VMS Software

VSI Data Dictionary (VDD) Installation and User Guide

Operating System and Version: VSI OpenVMS x86-64 Version 9.2-3 or higher
VSI OpenVMS IA-64 Version 8.4-2L1 or higher

Software Version: VSI Data Dictionary Version X1.2-03

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

VSI Data Dictionary (VDD) Installation and User Guide

Copyright © 2026 VMS Software, Inc. (VSI), Boston, Massachusetts, USA

Legal Notice

Confidential computer software. Valid license from VSI required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for VSI products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. VSI shall not be liable for technical or editorial errors or omissions contained herein.

All other trademarks and registered trademarks mentioned in this document are the property of their respective holders.

Table of Contents

Preface ..... **v**
1. About VSI v
2. Intended Audience v
3. Related Documents v
4. System Requirements v
5. Logging v
6. VSI Encourages Your Comments v
7. OpenVMS Documentation vi

Chapter 1. Installing VDD **1**
1.1. Downloading the PCSI Kit 1
1.2. Installing VDD 1
1.3. Uninstalling VDD 2
1.4. Linking With Legacy Executable 3
1.5. Included Utilities 3

Chapter 2. VDO Utility **5**
2.1. Commands 5
2.2. Scripts 5
2.3. Operations 5
2.3.1. Enabling CDDL-compatible Record Definition Semantics 5
2.3.2. Creating a New Repository 5
2.3.3. Showing the Default Directory 6
2.3.4. Changing the Current Directory 6
2.3.5. Listing Elements 6
2.3.6. Creating a New Directory 7
2.3.7. Deleting a Directory 7
2.3.8. Creating a New Field 7
2.3.9. Deleting a Field 7
2.3.10. Printing the Definition of an Element 8
2.3.11. Creating a New Record 8
2.3.12. Deleting a Record 8
2.3.13. Invoking HELP 9
2.3.14. Exiting VDO 9
2.4. Supported Element Types 9
2.5. Supported Nested Types 10
2.6. VDO Logging 10

Chapter 3. VMU Utility **11**
3.1. Prerequisites Check 11
3.2. VMU Logging 11
3.3. VMU Commands 12
3.4. VMU Examples 15

Chapter 4. VDMU Utility **17**
4.1. VDMU Commands 17

Chapter 5. VDD APIs **21**
5.1. The API Structure 21
5.2. VDD Buffers 21
5.3. Examples 21
5.3.1. Initialization and Clean-up 21

5.3.2. Dumping a Buffer 22
5.3.3. Creating a Directory 23
5.3.4. Deleting a Directory 24
5.3.5. Defining an Element 25
5.4. VDD Transactions 27
5.5. VDD Read-Only State 27
5.6. VDD ACL Buffers 27
5.7. VDD Element History 29
Chapter 6. Internal Architecture 31
6.1. The VDD Context 31
6.2. The VDD Process 31
6.3. VDD Helper Modules 32
Chapter 7. Using VDD With Layered Products 33
7.1. ACMS Applications With VDD 33
7.1.1. Building ACMS Examples 33
7.1.2. Prerequisites 33
7.1.3. ACMS Examples 34
7.1.4. Verifying ACMS Examples 35
7.2. TDMS With VDD 36
7.3. Datatrieve With VDD 36
7.3.1. Prerequisites 37
7.3.2. Setting Up Datatrieve 37
7.3.3. Testing Datatrieve 38
Chapter 8. Building VMS Applications 41
8.1. Building a BASIC Application 41
8.2. Building a FORTRAN Application 42

Preface

1. About VSI

VMS Software, Inc. (VSI) is an independent software company licensed by Hewlett Packard Enterprise to develop and support the OpenVMS operating system.

2. Intended Audience

This manual is intended for application programmers and system administrators who want to replace their existing Database Dictionaries with VDD or who want to take advantage of the new features provided by VDD. This manual contains administrative instructions as well as code samples.

3. Related Documents

The following documents provide additional useful information:

- *VSI C User Manual* [https://docs.vmssoftware.com/vsi-c-user-s-guide-for-openvms-systems/]
- *VSI OpenVMS System Manager's Manual, Volume 1: Essentials* [https://docs.vmssoftware.com/vsi-openvms-system-manager-s-manual-volume-1-essentials]
- *VSI OpenVMS System Manager's Manual, Volume 2: Tuning, Monitoring, and Complex Systems* [https://docs.vmssoftware.com/vsi-openvms-system-manager-s-manual-volume-2-tuning-monitoring-and-complex-systems]
- *VSI OpenVMS Programming Concepts Manual, Volume I* [https://docs.vmssoftware.com/vsi-openvms-programming-concepts-manual-volume-i]
- *VSI OpenVMS Programming Concepts Manual, Volume II* [https://docs.vmssoftware.com/vsi-openvms-programming-concepts-manual-volume-ii]

4. System Requirements

VDD is currently supported on VSI OpenVMS x86-64 Version 9.2-3 or higher and VSI OpenVMS IA-64 Version 8.4-2L1 or higher.

5. Logging

All VDD tools and libraries either implement built-in logging facilities, or utilize the default VDD logging facility. These log files can be useful for troubleshooting purposes and should be provided to VSI Support when reporting any issues.

6. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

7. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

Chapter 1. Installing VDD

Warning for Existing VDD Users

Users upgrading from a version of VDD prior to X1.2-02 must rebuild their repositories using the most recent version of VMU included in this kit.

1.1. Downloading the PCSI Kit

VDD is provided as a PCSI kit that includes all libraries and utilities required in order to use the software.

For detailed information on PCSI kits and working with them, refer to the *VSI OpenVMS POLYCENTER Software Installation Utility Manual* [https://docs.vmssoftware.com/vsi-openvms-polycenter-software-installation-utility-manual/].

1.2. Installing VDD

Follow this procedure:

1. To install the VDD PCSI kit, enter the following command:

   ```
   $ PRODUCT INSTALL VDD
   ```

   You will see output similar to the following:

   ```
   The following product has been selected:
       VSI X86VMS VDD X1.2-3                  Layered Product [Installed]

   Do you want to continue? [YES]

   Configuration phase starting ...

   You will be asked to choose options, if any, for each selected product and
    for
   any products that may be installed to satisfy software dependency
    requirements.

   Configuring VSI X86VMS VDD X1.2-3: Product: VSI X86VMS VDD X1.2-03 full

       Copyright 2026 VMS Software, Inc.  All rights reserved.

       Producer: VMS Software, Inc.
   ```

2. You will see the prompt `Do you want the defaults for all options?`

   If you answer **`YES`**, the procedure will automatically load ACMS system workspace definitions into your VDD repository. If you prefer to do this yourself, answer **`NO`**.

   The installation will then proceed as follows:

   ```
   Execution phase starting ...

   The following product will be installed to destination:
       VSI X86VMS VDD X1.2-3                  DISK$PIPPIN_SYS:[VMS$COMMON.]
   ```

```
Portion done: 0%...10%...50%...90%...100%

The following product has been installed:
    VSI X86VMS VDD X1.2-3                  Layered Product

VSI X86VMS VDD X1.2-3: Product: VSI X86VMS VDD X1.2-03 full

    Post-installation tasks are required:

    To use VDO and VDD users must execute
        @SYS$STARTUP:VDD$STARTUP.COM
        @SYS$MANAGER:VDD$SETUP_USER_ENV.COM
```

3. After the installation has completed, run the following script to set up the VDD system environment:

```
$ @SYS$STARTUP:VDD$STARTUP.COM
```

Note

The VDD$STARTUP file must be run as part of the system startup to make VDD operational.

---

4. Verify that the VDD$ANCHOR logical name has been defined by entering the following command:

```
$ SHOW LOGICAL VDD$ANCHOR
```

You should see output similar to the following:

```
(LNM$SYSTEM_TABLE)

  "VDD$ANCHOR" = "SYS$COMMON:[SYSTEMREPO]"
```

5. After you have set up the system environment, run the following script to set up the VDD user environment:

```
$ @SYS$MANAGER:VDD$SETUP_USER_ENV.COM
```

6. To verify that the user environment has been set up correctly, confirm that you have the required logical names in your process table and the required symbols defined. Enter the commands listed in the table below (in any order) and compare your outputs.

Command	Output
SHOW LOGICAL CDDSHR	"CDDSHR" = "SYS$LIBRARY:VDDSHR.EXE;" (LNM $PROCESS_TABLE)
SHOW LOGICAL CDDLIBSHR	"CDDLIBSHR" = "SYS$LIBRARY:VDDLIBSHR.EXE;" (LNM $PROCESS_TABLE)
SHOW SYMBOL VDO	VDO == "$VDO"
SHOW SYMBOL VMU[1]	VMU == "$VMU"
SHOW SYMBOL VDMU	VDMU == "$VDMU"
SHOW SYMBOL VDDL	VDDL :== "$VDO/DDL"

[1] IA-64 only

1.3. Uninstalling VDD

To uninstall VDD, enter the following command:

```
$ PRODUCT REMOVE VDD
```

This command will remove all VDD components, but it will *not* touch any user-created dictionaries (those created by users using the **DEFINE DICTIONARY** or **DEFINE REPOSITORY** commands).

1.4. Linking With Legacy Executable

As noted above, the logical name CDDSHR should be defined once you have run the VDD$SETUP_USER_ENV.COM file after installation. Layered products and custom applications that use CDD APIs should be able to transparently use the new VDD library, which exports the same set of symbols and implements equivalent runtime behavior.

1.5. Included Utilities

The installed kit includes the following utility tools:

VDO (VDD Operator)

A VDD operator utility, analogous with CDO.

VMU (VDD Migration Utility) – IA-64 Only

A CDD-to-VDD conversion tool for migrating legacy repositories.

VDMU (VSI Dictionary Management Utility)

A dictionary management utility used to manipulate dictionary directories.

Chapter 2. VDO Utility

The VDD Operator utility (VDO) is the main utility provided by VDD for interacting with VDD repositories. It is designed to replicate the functionality of the Oracle CDD CDO tool. Unlike CDO, VDO merges the legacy DDL syntax (as used by the VDMU/CDDL tools) with the newer CDO syntax. This has the advantage of supporting both repository types without dealing with extra logical names and dictionary files.

2.1. Commands

VDO implements its own parser, which aims to support all the commands and syntax needed to operate with VDD dictionaries. Currently, all VDO commands must be typed out in full (not abbreviated). All commands are case-insensitive.

There are also a number of dummy commands (no-op) that are designed to support legacy operations, and they have no real meaning within the context of VDD. The decision of keeping these no-op commands was made to maintain backwards compatibility with existing CDO scripts.

2.2. Scripts

Various VDO operations can be automated by using VDO script files, just as for the older CDO or CDDL tools. The VDO parser iterates over the tokens in such files without having a context of execution mode. While VDO does not differentiate between file extensions, VDO scripts typically use the .VDO file extension (for example, EXAMPLE.VDO).

2.3. Operations

This section details the operations that users can perform with the VDO utility.

2.3.1. Enabling CDDL-compatible Record Definition Semantics

Enabling CDDL-compatible record definition semantics in VDO allows you to define records both with and without nested STRUCTURE keywords.

To enable or disable this feature in an existing VDO session, enter **`SET DDL`** or **`SET NODDL`** respectively.

To start VDO with this feature enabled, enter **`VDDL`**.

2.3.2. Creating a New Repository

To create a new repository, you can either use the **`DEFINE REPOSITORY`** command or the legacy **`DEFINE DICTIONARY`** command. To specify where the repository should be created, supply the directory path after the command as follows:

```
VDO> DEFINE REPOSITORY [.VDD]
```

After the operation completes, you should see that the file REPOSITORY.DB has been created in the specified directory.

2.3.3. Showing the Default Directory

To show the current virtual directory where the VDO Utility operates, you can use the **SHOW DEFAULT** command:

```
VDO> SHOW DEFAULT
SYS$COMMON:[SYSTEMREPO]
```

2.3.4. Changing the Current Directory

When VDO is started, it tries to find the default repository file (REPOSITORY.DB) in the current working directory. If it finds the repository file, then it assumes that the current directory should be set for future use. In case of failure (when there is no REPOSITORY.DB in the current directory), VDO by default sets the current working directory to SYS$COMMON:[SYSTEMREPO]. Unlike the CDD, VDO does not maintain multiple logical names to locate dictionary files.

To set the current working directory, use the following VDO command:

```
VDO> SET DEFAULT DISK:[DIR]
```

Note

When setting default to a particular directory, VDO does not check whether the directory is valid or not; it simply stores the path for use by future VDO operations.

2.3.5. Listing Elements

To show which VDD records or elements are available, you can use the **DIRECTORY** command. The command accepts a *name* argument. However, if the argument is not provided, VDO lists all of the elements that are available in the current working directory:

```
VDO> DIRECTORY
Directory repository-root
CDD$PROTOCOLS                                   DIRECTORY
ACMS$DIR                                        DIRECTORY
F1(1)                                           FIELD
F1(3)                                           FIELD
F1(4)                                           FIELD
F1(5)                                           FIELD
REC1(1)                                         CDD$RECORD
REC1(2)                                         CDD$RECORD
```

Note

VDO will print objects in the order of their creation. The entity listing is sorted by creation time. VDO will print all versions of the entity that exist. Note that directories do not have versions.

The first line indicates the directory type. In the above example, *repository-root* indicates that we are currently in the root directory for the repository in question.

You can also show the contents of a specific directory as follows:

```
VDO> DIRECTORY ACMS$DIR.*
```

```
Directory ACMS$DIR

ACMS$WORKSPACES     DIRECTORY
```

2.3.6. Creating a New Directory

To create a new directory, you can use the **DEFINE DIRECTORY** command. This command can be used for both DDL- and CDO-style directories:

```
VDO> DEFINE DIRECTORY DIR.
```

2.3.7. Deleting a Directory

To delete the given directory, you can use the **DELETE DIRECTORY** command:

```
VDO> DELETE DIRECTORY DIR.
```

2.3.8. Creating a New Field

To define a new field, you can use the **DEFINE FIELD** command:

```
VDO> DEFINE FIELD FLD.
DEFINE FIELD FLD.
```

Note that after defining elements, VDO will print the complete element definition. The above example creates an "empty" field with no attributes. You can specify various attributes after specifying the field names. In the example below, you can see how the `DATATYPE` attribute is used:

```
VDO> DEFINE FIELD FLD DATATYPE IS WORD.
DEFINE FIELD FLD DATATYPE IS WORD.
```

2.3.9. Deleting a Field

To delete a field, you can use the **DELETE FIELD** command:

```
VDO> DELETE FIELD FLD.
```

You can also specify which version of a field will be deleted. In the following example, version `3` of the field `F1` is deleted:

```
VDO> DIRECTORY
Directory repository-root
CDD$PROTOCOLS                                   DIRECTORY
ACMS$DIR                                        DIRECTORY
F1(3)                                           FIELD
F1(1)                                           FIELD
F1(4)                                           FIELD
F1(5)                                           FIELD
REC1(1)                                         CDD$RECORD
REC1(2)                                         CDD$RECORD

! DELETE A FIELD
VDO> DELETE FIELD F1;3.
VDO> DIRECTORY
Directory repository-root
CDD$PROTOCOLS                                   DIRECTORY
```

```
ACMS$DIR                                                DIRECTORY
F1(1)                                                   FIELD
F1(4)                                                   FIELD
F1(5)                                                   FIELD
REC1(1)                                                 CDD$RECORD
REC1(2)                                                 CDD$RECORD
```

If you do not specify any version, or leave a trailing semicolon (;) at the end of the field name, the latest version of the field will be deleted.

2.3.10. Printing the Definition of an Element

You can use the **EXTRACT *name*** command to print out the definition of an element, as illustrated in the following example:

```
VDO> EXTRACT SAMPLE$ROOT.SAMPLE$RECORD

DEFINE RECORD SAMPLE$ROOT.SAMPLE$RECORD.
   SAMPLE$PROCESSING_STATUS STRUCTURE .
      SAMPLE$L_F0 DATATYPE IS LONGWORD INITIAL_VALUE IS 1.
      SAMPLE$T_F1 DATATYPE IS TEXT SIZE IS 1 CHARACTERS INITIAL_VALUE IS "S".
      SAMPLE$T_F2 DATATYPE IS TEXT SIZE IS 1 CHARACTERS INITIAL_VALUE IS "G".

      VARIANTS.
        VARIANT.
           SAMPLE$T_V0 DATATYPE IS TEXT SIZE IS 132 CHARACTERS INITIAL_VALUE IS " ".
        END VARIANT.
        VARIANT.
           SAMPLE$T_V1 DATATYPE IS TEXT SIZE IS 80 CHARACTERS.
        END VARIANT.
     END VARIANTS.
   END STRUCTURE.
END RECORD.
```

As you can see, the entire definition is printed from top to bottom.

2.3.11. Creating a New Record

To create a new record, you can use the **DEFINE RECORD** command. Note that the definition should end with its respective **END RECORD** command:

```
VDO> DEFINE RECORD RCD. END RECORD.
DEFINE RECORD RCD.
END RECORD.
```

Records generally contain a body, which itself is a collection of fields, arrays, or other record types. You can also nest records.

2.3.12. Deleting a Record

To delete a record, you can use the **DELETE RECORD** command:

```
VDO> DELETE RECORD RCD.
```

You can also specify which version of a record will be deleted. In the following example, version `1` of the record `REC1` is deleted:

```
VDO> DIRECTORY
Directory repository-root
```

```
CDD$PROTOCOLS                                                   DIRECTORY
ACMS$DIR                                                        DIRECTORY
F1(1)                                                           FIELD
F1(4)                                                           FIELD
F1(5)                                                           FIELD
REC1(1)                                                         CDD$RECORD
REC1(2)                                                         CDD$RECORD

!DELETE A RECORD
VDO> DELETE RECORD REC1;1.
VDO> DIRECTORY
Directory repository-root
CDD$PROTOCOLS                                                   DIRECTORY
ACMS$DIR                                                        DIRECTORY
F1(1)                                                           FIELD
F1(4)                                                           FIELD
F1(5)                                                           FIELD
REC1(2)                                                         CDD$RECORD
```

2.3.13. Invoking HELP

You can enter the VDO **HELP** command to display helpful information about the VDO utility:

```
$ VDO
VDO> HELP

  Information available:

  Expressions           File_Area_Key         fld-properties
  Logical_Names         rec-properties        Start-up-info
  Sys-properties        User-properties       VDO_CommandProc
  VDO_Commands          VDO_Edit_Strs

Topic?
```

2.3.14. Exiting VDO

To exit the VDO utility, you can either enter the **EXIT** command or use the **Ctrl/Z** key binding:

```
$ VDO
VDO> EXIT
$ VDO
VDO> Ctrl/Z
$
```

2.4. Supported Element Types

VDO currently supports the following Element Types:

- Directory
- Field
- Record (both legacy and new versions of it)
- ACMS Task

- ACMS Task Group
- ACMS Application

Note

The ACMS$MENU entity type is still in development.

To create more complex data types (such as those required by ACMS), use the **GENERIC** keyword before the actual element type, as illustrated below:

```
VDO> DEFINE GENERIC ACMS$TASK T.
cont.> END GENERIC.

…
VDO> DEFINE GENERIC ACMS$TASK_GROUP G.
cont.> END GENERIC.
```

Note

ACMS and Datatrieve elements cannot be mixed with other types within nested structures, meaning that you cannot create an ACMS Task and have regular element types embedded within it, or vice versa.

2.5. Supported Nested Types

As noted previously, you can create nested elements within a record. These elements can have a regular type, but they also can be one of the following types:

- Overlay
- Structure
- Array

2.6. VDO Logging

VDO does not have a separate logging facility like other VDD utilities. It uses the default logging mechanism provided by VDD, which will generate log files with the default name of VDD.LOG. These log files can usually be found in the current working directory.

Note

You must define `VDD_LOG_LEVEL` *`log-level`* in order to enable the VDD logging facility for VDO.

Chapter 3. VMU Utility

The VDD Migration Utility (VMU) is a CDD-to-VDD conversion tool for migrating legacy repositories on IA-64 systems.

The following prerequisites must be met before migrating using the VMU utility:

- You are using VSI OpenVMS IA-64 Version 8.4 or higher.
- You have a properly installed and configured Oracle CDD/Repository or CDD/Plus.
- You have a properly installed and configured version of the most recent VSI VDD kit.
- Your VDD dictionary is from the most recent VSI VDD kit release.
- Your Oracle CDD/Repository or CDD/Plus dictionary is filled up by any method (i.e. dump or manually).

Additionally, be aware of the following usage requirements:

- VDD *must* be prepared using its startup script (i.e. @SYS$STARTUP:VDD$STARTUP.COM).
- The CDDSHR logical name *must not* be defined, or should lead to the Oracle CDD/Repository or the CDD/Plus main shared library.
- The CDDLIBSHR logical name *must not* be defined, or should lead to the Oracle CDD/Repository or the CDD/Plus main shared library.
- The VDD$DEFAULT logical name *must not* be defined.
- The CDD$DEFAULT logical name *must not* be defined.
- The VDD$ANCHOR logical name *must* lead to the destination directory.

3.1. Prerequisites Check

The VMU tool supports prerequisite checks embedded in the tool itself. The content requirements of the command line of the VMU tool (described in detail above) are checked before any instance of migration or analysis is invoked.

The following example shows the message that will appear if a requirement is not satisfied:

```
$ VMU MIGRATE CDD$TOP.TDMS$SAMPLES
%VMU-F-REQNOTMET, VMU Prerequisites are not met, please refer to VMU HELP
$
```

3.2. VMU Logging

The VMU tool features its own logging mechanism. Log messages are written to VMU.LOG, and this file can be found in your current DCL default directory.

To activate logging for VMU, the VMU_LOG_LEVEL logical name must be present in the logical name table.

There are a total of four log levels:

- Debug level ("DBG")
- Informational level ("INF")
- Warning level ("WRN")
- Error level ("ERR")

After invocation of the VMU migration or analysis instance, a log file will automatically be created.

3.3. VMU Commands

The VMU tool now supports commands and qualifiers, bringing it in line with other OpenVMS tools and making it more user-friendly for OpenVMS systems.

This section details the VMU commands that are available in the current release.

VMU MIGRATE

VMU MIGRATE — Use the **VMU MIGRATE** command to migrate data from the CDD database directory to a VDD dictionary. The migration process may optionally appear with live update of the migration material, including the processed nodes, their attributes, and overall coverage of the migration.

Format

VMU MIGRATE (CDD$TOP.*CDD_BASE*) [CDD$TOP.*VDD_BASE*]

Parameters

(*CDD_BASE*) – Required

The directory source of the CDD dictionary that will be migrated. The *CDD_BASE* parameter *must* be preceded by the CDD$TOP prefix.

[*VDD_BASE*] – Optional

The directory target of the VDD dictionary where the migration material will be migrated to. If specified, the *VDD_BASE* parameter *must* be preceded by the CDD$TOP prefix. If the *VDD_BASE* parameter is not specified, the target base will be assumed the same as *CDD_BASE*, and you will see the following message in the console:

```
%VMU-I-VDD_MISS, VDD_BASE Parameter is not supplied, replicating CDD_BASE
```

Qualifiers

/[NO]LIVE - Optional

Use the **/LIVE** qualifier with the **VMU MIGRATE** command to force live output during the migration.

To negate this configuration, use the **/NOLIVE** qualifier. With **/NOLIVE** enabled, you will not see live updating statistics of the migration material in the console. Only messages will be displayed alongside the migration.

The default is **/LIVE**.

/RESUME - Optional

Use this qualifier with the **VMU MIGRATE** command to resume the most recently aborted migration. The resume point will store the last processed entity from the CDD database in the logical name VMU$$RESUME_POINT in the SYSTEM logical name table.

If the VMU$$RESUME_POINT logical name was not deleted from the server's SYSTEM logical name table, then you will see following informational message in the console:

```
%VMU-I-RESUMEFOUND, Resume point was caught, resuming migration...
```

Otherwise, if this qualifier is supplied, but the logical name was not present in server's logical name table, you will see the following warning message in the console:

```
%VMU-W-RESUMENOTFND, Resume point is not available, starting from CDD_BASE
```

The VMU$$RESUME_POINT logical name will automatically be deleted upon successful migration.

/[NO]VERSION - Optional

Use the **/VERSION** qualifier with the **VMU MIGRATE** command to migrate all versions of entities in the specified CDD dictionary. Be aware that specifying the **/VERSION** qualifier will increase processing time and disk usage of the migration.

If the **/NOVERSION** qualifier is specified, only the latest version will be migrated.

The default is **/NOVERSION**.

VMU ANALYZE

VMU ANALYZE — Use the **VMU ANALYZE** command if you want to analyze the migration material from CDD Dictionary. Currently, the **VMU ANALYZE** command can traverse the migration material sub-tree and display the statistics of the supplied directory. Additional features are planned for the future.

Format

VMU ANALYZE (CDD$TOP.*CDD_BASE*)

Parameters

(*CDD_BASE*) - Required

The directory source of the material that will be analyzed. The path of the directory *must* be preceded by the CDD$TOP prefix.

Qualifiers

/[NO]LIVE - Optional

Use the **/LIVE** qualifier with the **VMU ANALYZE** command to force live output during the analysis.

To negate this configuration, use the **/NOLIVE** qualifier. With **/NOLIVE** enabled you will not see live updating statistics of the analysis material in the console. With this configuration, only messages will be displayed alongside the analysis.

The default is **/LIVE**.

/STATISTICS - Optional

This qualifier is always assumed to be passed, as the statistics analysis is the only metric that VMU can gather. If you use the **VMU ANALYZE** command but do not specify this qualifier, you will see the following informational message in the console:

```
%VMU-I-STAT_MISS, /STATISTICS qualifier missing, implicitly assuming /STATISTICS.
```

VMU HELP

VMU HELP — Use the **VMU HELP** command to display short usage hints for the VMU tool.

Format

VMU HELP

Example

```
$ VMU HELP

VMU

        VMU is a VDD Migration Utility which is a part of VSI VDD toolkit.

        VDD migration utility is used to migrate currently active DMU (part
        of Oracle CDD/Repository or CDD/Plus toolkit) dictionary to VDD
        dictionary using "Old API".

        Prerequisites:

                o Architecture: IA64 Only
                o Operating system: OpenVMS v8.4
                o Properly installed and configured Oracle CDD/Repository or
                  CDD/Plus
                o Properly installed and configured VSI VDD
                o Oracle CDD/Repository or CDD/Plus dictionary filled up by any
                  method (i.e. dump or manually)

        Usage:

                o VDD MUST be prepared using its startup script
                  (i.e. @SYS$STARTUP:VDD$STARTUP.COM)

                o CDDSHR logical name MUST NOT be defined or should lead to
                  Oracle CDD/Repository or CDD/Plus main shared library

                o CDDLIBSHR logical name MUST NOT be defined or should lead to
                  Oracle CDD/Repository or CDD/Plus main shared library

                o VDD$DEFAULT logical name MUST NOT be defined

                o CDD$DEFAULT logical name MUST NOT be defined
```

```
                o VDD$ANCHOR MUST lead to repository where migration to be
                  provided

  Additional information available:

  MIGRATE    ANALYZE    HELP       Examples
```

3.4. VMU Examples

Example 3.1. Migrating a Directory

The following example shows a migration of the TDMS$SAMPLES directory from a CDD dictionary to a VDD dictionary using the VMU tool:

```
$ VMU MIGRATE CDD$TOP.TDMS$SAMPLES CDD$TOP.SAMPLE$DB

Processed:

    directories:               2
    current directory:         _CDD$TOP.TDMS$SAMPLES.DEPARTMENT

    terminal nodes:            2
    current terminal node:     _CDD$TOP.TDMS$SAMPLES.DEPARTMENT.
DEPTAPSCH_REQUESTRD

    entities:                  28
    attributes overall:        72
    attributes by types:
        ENTITY:                12
        ENTITY LIST:           9
        NULL:                  0
        NUMERIC:               34
        STRING:                16
        STRING LIST:           1
```

Example 3.2. Resuming Migrating a Directory

In the following example, we assume that a migration has been aborted during execution, and we want to resume from where it was stopped. We use the **/RESUME** qualifier to tell VMU take into account the previously aborted migration:

```
...
! MIGRATION WAS ABORTED!
...
$ SHOW LOG VMU$$RESUME_POINT
   "VMU$$RESUME_POINT" = "TDMS$SAMPLES.DEPARTMENT. DEPTAPUPD_REQUEST"
(LNM$SYSTEM_TABLE)
$
$ VMU MIGRATE/RESUME CDD$TOP.TDMS$SAMPLES
%VMU-I-VDD_MISS, VDD_BASE Parameter is not supplied, replicating CDD_BASE
%VMU-I-RESUMEFOUND, Resume point was caught, resuming migration...

Processed:

    directories:               2
    current directory:         _CDD$TOP.TDMS$SAMPLES.DEPARTMENT

    terminal nodes:            4
    current terminal node:     _CDD$TOP.TDMS$SAMPLES.DEPARTMENT.
DEPTAPUPD_REQUESTCE
```

```
    entities:                   327
    attributes overall:         673
    attributes by types:
        ENTITY:                 173
        ENTITY LIST:            107
        NULL:                   3
        NUMERIC:                239
        STRING:                 145
        STRING LIST:            6
```

Example 3.3. Gathering Analytics

The VMU tool allows users to gather analytics of the migration material without actually migrating anything to VDD. The following example procedure shows statistics of the specified source directory in a CDD dictionary:

```
$ VMU ANALYZE/STATISTICS CDD$TOP.TDMS$SAMPLES

Processed:

    directories:                6
    current directory:

    terminal nodes:             127
    current terminal node:
STAT_WORKSPACESTFORM

    entities:                   8564
    attributes overall:         18752
    attributes by types:
        ENTITY:                 4507
        ENTITY LIST:            1664
        NULL:                   139
        NUMERIC:                5107
        STRING:                 7103
        STRING LIST:            232
Directory CDD$TOP.TDMS$SAMPLES done analysis.
$
```

Chapter 4. VDMU Utility

The VSI Dictionary Management Utility (VDMU) is intended to manipulate the legacy part of VDD and dictionary directories. This utility is used in various layered products to prepare dictionary models before the layered products can utilize the dictionary.

4.1. VDMU Commands

This section details the commands that are supported in VDMU.

BACKUP (Experimental)

BACKUP (Experimental) — Use the **BACKUP** command to copy the directory hierarchy and related data definitions into a backup file. This allows you to keep a backup copy of your directory hierarchy, complete with history lists and access control lists.

Format

BACKUP [*qualifiers*] [*path-name* [, *path-name*]]... *file-specification*

Qualifiers

/[NO]PROTECTION

Use **/PROTECTION** to include access control lists in the backup file. Use **/NOPROTECTION** to exclude access control lists from the information copied into the backup file.

The default is **/NOPROTECTION**.

CREATE

CREATE — Use the **CREATE** command to create dictionary directories. Be aware that VDMU **CREATE** does not create Terminal entities and VDD objects—this command is only for creating directories. You can also create access control lists and history lists for the new directories. If any ancestors of the directory to be created do not exist, then VDD creates them automatically.

Format

CREATE [*qualifiers*] *path-name*]

Qualifiers

/AUDIT [= (*quoted-string* [, *quoted-string*]...)]
/AUDIT=*file-specification*
/NOAUDIT

Use **/AUDIT** to create a history list entry auditing the creation of a dictionary directory.

You can include explanatory text in history list entries by including quoted strings. Enclose each quoted string in double quotation marks, and enclose the series of strings in parentheses. The parentheses are optional if you specify only one quoted string.

With **/NOAUDIT**, no history list entries are created.

The default is **/NOAUDIT**.

DELETE

DELETE — Use the **DELETE** command to delete dictionary directories, subdictionaries, and objects. As the children of a dictionary directory or subdictionary are always deleted with their parent, the command has qualifiers to determine whether directories or subdictionaries with children should be deleted.

Format

DELETE [*qualifiers*] *path-name* [, *path-name*]...

Qualifiers

/ALL

Use **/ALL** to delete the specified dictionary directory or subdictionary and all its descendants.

/[NO]LOG

Use **/LOG** to create a list of the given names of the dictionary directories and objects deleted. Note that, if you specify **/ALL** when deleting a dictionary directory with children, **/LOG** does not list the names of the deleted children.

The default is **/NOLOG**.

EXIT

EXIT — Use the **EXIT** command or **Ctrl/Z** to return to DCL command level.

Format

EXIT

or

Ctrl/Z

HELP

HELP — Use the **HELP** command to display additional information about VDMU commands.

Format

HELP

Example

```
VDMU> HELP

VDMU
```

```
  Additional information available:

  BACKUP      CREATE      DELETE      EXIT        HELP        LIST        SET
  SHOW        specify

VDMU Subtopic?
```

LIST

LIST — Use the **LIST** command to display information about dictionary directories, subdictionaries, and objects. This information includes sources, history lists, access control lists, and text.

Format

LIST

Example

```
VDMU> LIST
        A
        ACCOUNT_RECORD;1 <CDD$DATA_AGGREGATE>
        CDD$PROTOCOLS
        INSPECT_DATA;1 <CDD$DATA_ELEMENT>
        NAME_LENGTH;1 <CDD$DATA_ELEMENT>
        RECORD_SIZE;1 <CDD$DATA_ELEMENT>

VDMU>
```

SET

SET — Currently, the **SET** command only supports the **DEFAULT** command. Use the **SET DEFAULT** command to temporarily set your default CDD directory. This directory will become your default as you proceed. You cannot use wildcards in the path name. If you are using a terminal of the VT200 family, you can use 8-bit characters in path names.

Format

SET DEFAULT *path-name*

SHOW

SHOW — Currently, the **SHOW** command only supports the **DEFAULT** command. Use the **SHOW DEFAULT** command to display your current default CDD directory.

Format

SHOW DEFAULT

Chapter 5. VDD APIs

To automate VDD operations, you can use the VDO script files as discussed previously. However, the entire functionality of VDD is not exposed via VDO commands. To be able to use the extended features of VDD, it is possible to use VDD's APIs, which are a set of functions provided by the VDD shareable image that can be used to perform various VDD operations programmatically.

5.1. The API Structure

The shared binary includes all the CDD symbols, except those from the *MCS_* API. VDD provides header files for the constants such as the enumeration types or numeric literals used by the programmable interface. However, the APIs themselves are not currently predefined in the header files.

5.2. VDD Buffers

Some of the API functions require programs to pass a VDD buffer, which consists of descriptions of particular procedure behavior. This was done to eliminate the need for passing a large amount of arguments into such functions. This also maintains compatibility with the CDD procedures.

5.3. Examples

The following examples illustrate how to use the APIs. Note that while all examples are written in C, the APIs themselves are largely language-independent.

5.3.1. Initialization and Clean-up

There are multiple ways to initialize a VDD context. The legacy method is to use the SIGN_IN/OUT methods, whereas the newer method expects the user to create a context and then attach to it.

The following example shows the legacy method to initialize a VDD context:

```
long status;
long context;

status = vdd$sign_in(&context);
if (!(status & 1)) {
    lib$signal(status);
}
// ...

status = vdd$sign_out(&context);
```

The code simply initializes the context, which can then be used by the other procedures.

The following example shows how to initialize a VDD context with newer procedures:

```
long status;

long message_vector[20];
long nad_user_handle[2];
long context_handle[2];
```

```
ots$move5(0, NULL, 0, sizeof (message_vector), message_vector);
ots$move5(0, NULL, 0, sizeof (nad_user_handle), nad_user_handle);
ots$move5(0, NULL, 0, sizeof (context_handle), context_handle);

status = vdd$attach(message_vector, nad_user_handle, context_handle);
if (!(status & 1)) {
        sys$putmsg(message_vector);
        goto $CLEANUP:
}

long session_handle[2];

ots$move5(0, NULL, 0, sizeof (session_handle), session_handle);

status = vdd$start_session(message_vector, nad_user_handle,
                        session_handle, NULL, NULL, context_handle);
if (!(status & 1)) {
        sys$putmsg(message_vector);
        goto $CLEANUP:
}

// define $CLEANUP somewhere

status = vdd$end_session(message_vector, session_handle,
                        NULL, context_handle, NULL);
if (!(status & 1)) {
        sys$putmsg(message_vector);
}

status = vdd$detach(message_vector, nad_user_handle, context_handle);
if (!(status & 1)) {
        sys$putmsg(message_vector);
}
```

While the newer method is more verbose, it lets you control the behavior of the operations more precisely. Note that all procedures accept *message_vector* as an argument. This vector can be used to trace the errors. If you simply print out the message text associated with the *status* variable, this will only display the last error; with *message_vector* you have the entire error trace in one array.

On line 11, the above example first tries to attach to the VDD context. The supplied arguments are internal memory pointers that need to be passed to the procedures that perform VDD operations. The same applies to the procedure that is being used on line 21.

5.3.2. Dumping a Buffer

Sometimes, it can be useful to dump the VDD Buffer to see the errors. VDD APIs that use these buffers usually report errors when they are unable to parse the buffer. However, the error description is often not sufficient, which is why we have to rely on a debugging APIs such as the VDD$DUMP_BUFFER to obtain additional information when a problem is encountered:

```
// assuming you have the buffer built before calling this procedure

unsigned short buffer_length;
vdd$dump_buffer(msg_vec, &buffer_length, buffer, session);

// OUTPUT:
// 1  begin
// 13 directory_info_dsc V3.0
// 14    directory_name_list
// 56       directory_name "DISK:[REPO]CDD$PROTOCOLS"
```

```
// 57    end
// 62    type MCS_DIRECTORY
// 63 eoc
```

This VDD debug API call parses the binary data associated with the operation in question and prints it with VDD tokens. In the case of errors, you will find out immediately which portion is corrupted. The listed line numbers indicate the offset in that given buffer, where the names are mnemonics of VDD Tokens.

5.3.3. Creating a Directory

To create a directory, you can use the VDD$CREATE_DIRECTORY procedure:

```
// initialize the VDD Context first
// ...

const uint8_t directory_path[] = "TEST$PARENT.TEST$CHILD";

uint8_t directory_buffer[64];
uint8_t *directory_pointer;
{
    directory_pointer = directory_buffer;

    *(uint8_t*)directory_pointer = vdd$k_begin;
    directory_pointer += sizeof (uint8_t);
    *(uint32_t*)directory_pointer = vdd$k_directory_info_dsc;
    directory_pointer += sizeof (uint32_t);
    *(uint32_t*)directory_pointer = 2;
    directory_pointer += sizeof (uint32_t);
    *(uint32_t*)directory_pointer = 0;
    directory_pointer += sizeof (uint32_t);
    {
        *(uint8_t*)directory_pointer = vdd$k_directory_name_list;
        directory_pointer += sizeof (uint8_t);
        {
            *(uint8_t*)directory_pointer = vdd$k_directory_name;
            directory_pointer += sizeof (uint8_t);
            *(uint16_t*)directory_pointer = sizeof (directory_path) - 1;
            directory_pointer += sizeof (uint16_t);
            ots$move3(sizeof (directory_path) - 1,
                    directory_path, directory_pointer);
            directory_pointer += sizeof (directory_path) - 1;
        }
        *(uint8_t*)directory_pointer = vdd$k_end;
        directory_pointer += sizeof (uint8_t);
    }
    *(uint8_t*)directory_pointer = vdd$k_eoc;
    directory_pointer += sizeof (uint8_t);
}

struct dsc$descriptor_s directory_descriptor;
directory_descriptor.dsc$w_length  = diectory_pointer - directory_buffer;
directory_descriptor.dsc$a_pointer = directory_buffer;
directory_descriptor.dsc$b_class   = DSC$K_CLASS_D;
directory_descriptor.dsc$b_dtype   = DSC$K_DTYPE_T;

long status;
status = vdd$create_directory(message_vector, session_handle,
                              &directory_descriptor);
if (!(status & 1)) {
    sys$putmsg(message_vector);
    goto $CLEANUP;
}
```

Here we can see how a VDD Buffer gets used. We first allocate a small portion of space, then we move around the buffer using an alias pointer. Each time we set a piece of the memory to some token we move the pointer forward. Later on, we use that pointer to calculate the final length.

The expressions in the buffer usually have a header and footer (see `k_begin` and `k_end` or `k_eoc` tokens). Inside of these expressions, you can specify the body of the expression.

5.3.4. Deleting a Directory

To delete a given directory, you can use the VDD$DELETE_DIRECTORY procedure:

```
// initialize the VDD Context first
// ...

const uint8_t directory_path[] = "TEST$PARENT.TEST$CHILD";

uint8_t directory_buffer[64];
uint8_t *directory_pointer;
{
    directory_pointer = directory_buffer;

    *(uint8_t*)directory_pointer = vdd$k_begin;
    directory_pointer += sizeof (uint8_t);
    *(uint32_t*)directory_pointer = vdd$k_directory_info_dsc;
    directory_pointer += sizeof (uint32_t);
    *(uint32_t*)directory_pointer = 2;
    directory_pointer += sizeof (uint32_t);
    *(uint32_t*)directory_pointer = 0;
    directory_pointer += sizeof (uint32_t);
    {
        *(uint8_t*)directory_pointer = vdd$k_directory_name_list;
        directory_pointer += sizeof (uint8_t);
        {
            *(uint8_t*)directory_pointer = vdd$k_directory_name;
            directory_pointer += sizeof (uint8_t);
            *(uint16_t*)directory_pointer = sizeof (directory_path) - 1;
            directory_pointer += sizeof (uint16_t);
            ots$move3(sizeof (directory_path) - 1,
                    directory_path, directory_pointer);
            directory_pointer += sizeof (directory_path) - 1;
        }
        *(uint8_t*)directory_pointer = vdd$k_end;
        directory_pointer += sizeof (uint8_t);
    }
    *(uint8_t*)directory_pointer = vdd$k_eoc;
    directory_pointer += sizeof (uint8_t);
}

struct dsc$descriptor_s directory_descriptor;
directory_descriptor.dsc$w_length  = diectory_pointer - directory_buffer;
directory_descriptor.dsc$a_pointer = directory_buffer;
directory_descriptor.dsc$b_class   = DSC$K_CLASS_D;
directory_descriptor.dsc$b_dtype   = DSC$K_DTYPE_T;

long status;
status = vdd$delete_directory(message_vector, session_handle,
                              &directory_descriptor);
if (!(status & 1)) {
    sys$putmsg(message_vector);
    goto $CLEANUP;
}
```

The code is similar to the previous example in terms of general structure.

5.3.5. Defining an Element

To define an element, we first need to build the desired VDD Buffer and then supply it to the VDD$DEFINE_ELEMENT procedure:

```
const char field_path[] = "DISK:[REPO]FIELD_0";

char el_buff[512];
char *el_ptr;
{
    el_ptr = el_buff;

    *(uint8_t*)el_ptr = vdd$k_begin;
    el_ptr += sizeof (uint8_t);
    *(uint32_t*)el_ptr = vdd$k_metadata_buf_dsc;
    el_ptr += sizeof (uint32_t);
    *(uint32_t*)el_ptr = 2;
    el_ptr += sizeof (uint32_t);
    *(uint32_t*)el_ptr = 0;
    el_ptr += sizeof (uint32_t);
    {
        *(uint8_t*)el_ptr = vdd$k_entity;
        el_ptr += sizeof (uint8_t);
        *(uint32_t*)el_ptr = vdd$k_ent_data_element;
        el_ptr += sizeof (uint32_t);
        *(uint32_t*)el_ptr = 1;
        el_ptr += sizeof (uint32_t);
        *(uint32_t*)el_ptr = 0;
        el_ptr += sizeof (uint32_t);
    }
    {
        *(uint8_t*)el_ptr = vdd$k_begin;
        el_ptr += sizeof (uint8_t);
        *(uint32_t*)el_ptr = vdd$k_directory_info_dsc;
        el_ptr += sizeof (uint32_t);
        *(uint32_t*)el_ptr = 2;
        el_ptr += sizeof (uint32_t);
        *(uint32_t*)el_ptr = 0;
        el_ptr += sizeof (uint32_t);
        {
            *(uint8_t*)el_ptr = vdd$k_directory_name_list;
            el_ptr += sizeof (uint8_t);
            {
                *(uint8_t*)el_ptr = vdd$k_directory_name;
                el_ptr += sizeof (uint8_t);
                *(uint16_t*)el_ptr = sizeof (field_path) - 1;
                el_ptr += sizeof (uint16_t);
                ots$move3(sizeof (field_path) - 1, field_path, el_ptr);
                el_ptr += sizeof (field_path) - 1;
            }
            *(uint8_t*)el_ptr = vdd$k_end;
            el_ptr += sizeof (uint8_t);
        }
        {
            *(uint8_t*)el_ptr = vdd$k_type;
            el_ptr += sizeof (uint8_t);
            *(uint32_t*)el_ptr = vdd$k_ent_element;
            el_ptr += sizeof (uint32_t);
        }
        *(uint8_t*)el_ptr = vdd$k_eoc;
        el_ptr += sizeof (uint8_t);
    }
    {
        *(uint8_t*)el_ptr = vdd$k_attribute_list;
        el_ptr += sizeof (uint8_t);
```

```
        {
            *(uint8_t*)el_ptr = vdd$k_attribute;
            el_ptr += sizeof (uint8_t);

            *(uint32_t*)el_ptr = vdd$k_att_processing_name;
            el_ptr += sizeof (uint32_t);
            *(uint8_t*)el_ptr = vdd$k_literal;
            el_ptr += sizeof (uint8_t);
            *(uint8_t*)el_ptr = DSC$K_DTYPE_T;
            el_ptr += sizeof (uint8_t);
            *(uint16_t*)el_ptr = sizeof (field_path) - 1;
            el_ptr += sizeof (uint16_t);
            ots$move3(sizeof (field_path) - 1, field_path, el_ptr);
            el_ptr += sizeof (field_path) - 1;
        }
        *(uint8_t*)el_ptr = vdd$k_end;
        el_ptr += sizeof (uint8_t);
    }
    *(uint8_t*)el_ptr = vdd$k_end;
    el_ptr += sizeof (uint8_t);
    *(uint8_t*)el_ptr = vdd$k_eoc;
    el_ptr += sizeof (uint8_t);
}
unsigned short el_diff;
el_diff = el_ptr - el_buff;

struct dsc$descriptor_s el_desc;
el_desc.dsc$a_pointer  = el_buff;
el_desc.dsc$w_length   = el_diff;
el_desc.dsc$b_class    = DSC$K_CLASS_D;
el_desc.dsc$b_dtype    = DSC$K_DTYPE_T;

long element_handle[2];
ots$move5(0, NULL, 0, sizeof (element_handle), element_handle);

long status;
status = vdd$define_element(msg_vec, session,
                            &el_desc, element_handle);
if (!(vdd_res & 1)) {
 lib$signal(status);
        return 1;
}
```

This example uses the `metadata_buf_dsc` token to include the element definition in it. First, we specify that our element is a field by using the `ent_data_element` token, and then we specify the path by using the `directory_info_dsc` token.

Having these two required descriptions, we can then start specifying the attributes of the element. We do this using the `attribute_list` token, which contains all of the attributes that should describe our new element. Here, we only specify one attribute, which is its name. We do this by embedding the `att_processing_name` into our buffer.

To specify values in the buffer, we often need to use the `literal` token. With this token we can precisely describe what type of value we want to have. In our case, we use a static character array, which is why we chose the default VMS `DSC$K_DTYPE_T` enum here. Once the buffer is complete, we can supply it to the procedure, which can be seen on line 97.

This procedure can be used to define all types of elements. You can define records, generic types such as the ACMS Task, or DATATRIEVE Objects. However, note that we cannot define a directory with this API call. This seeming inconsistency derives from equivalent CDD interface, which does not support such functionality.

5.4. VDD Transactions

VDD supports transactions via the API calls. Transactions can only be used in newer versions of API procedures, as illustrated below. To start a transaction, you need to set the transaction bit when calling the VDD$START_SESSION:

```
long transaction_flag;
transaction_flag = 1;

status = vdd$start_session(message_vector, nad_user_handle,
                           session_handle, NULL,
                           &transaction_flag, context_handle);
```

This will start the VDD Transaction within a session. Each time an operation is made by calling a VDD procedure, the internal context will keep the changes in a vector. In the case of a failure, the vector will be used to undo every single change that was done during the session.

Operations must check the last status in order to make sure that the last operation was completed successfully.

Additionally, there is the abort_flag, which can be used in the VDD$END_SESSION procedure to roll back all the changes that were made during the session:

```
long abort_flag;
abort_flag = 1;

status = vdd$end_session(message_vector, nad_user_handle,
                           &abort_flag, context_handle);
```

This flag only gets checked by the VDD$END_SESSION API call

5.5. VDD Read-Only State

In case of needing to only read data from the VDD database, the user can create a read-only session. To enter into the read-only state, you need to set the appropriate bit when calling the VDD$START_SESSION:

```
long readonly_flag;
readonly_flag = 1;

status = vdd$start_session(message_vector, nad_user_handle,
                           session_handle, &readonly_flag,
                           NULL, context_handle);
```

Here, VDD procedures will simply deny any attempt of making a write call to the database. Both *VDD Transactions* and *VDD Read-only* bits can be set simultaneously.

5.6. VDD ACL Buffers

VDD Elements can be protected via Access Control Entries. These can be set by using the VDD$DEFINE_ELEMENT or VDD$CHANGE_ELEMENT procedures. To supply extra information for these API calls, it is necessary to build an ACL buffer and attach it to a regular buffer:

```
unsigned char ch_buff[256];
unsigned char *ch_ptr;
{
    ch_ptr = ch_buff;
```

```
    *(uint8_t*)ch_ptr = vdd$k_begin;
    ch_ptr += sizeof (uint8_t);
    *(uint32_t*)ch_ptr = vdd$k_metadata_change;
    ch_ptr += sizeof (uint32_t);
    *(uint32_t*)ch_ptr = 2;
    ch_ptr += sizeof (uint32_t);
    *(uint32_t*)ch_ptr = 0;
    ch_ptr += sizeof (uint32_t);
    {
        *(uint8_t*)ch_ptr = vdd$k_attribute_change;
        ch_ptr += sizeof (uint8_t);
        {
            *(uint8_t*)ch_ptr = vdd$k_attribute_modify;
            ch_ptr += sizeof (uint8_t);

            *(uint32_t*)ch_ptr = vdd$k_att_acl;
            ch_ptr += sizeof (uint32_t);
            *(uint8_t*)ch_ptr = vdd$k_literal;
            ch_ptr += sizeof (uint8_t);
            *(uint8_t*)ch_ptr = vdd$k_dtype_unstructured;
            ch_ptr += sizeof (uint8_t);
            // ACL buffer length + the acl_dsc header length
            *(uint32_t*)ch_ptr = 12 + 14;
            ch_ptr += sizeof (uint32_t);

            *(uint8_t*)ch_ptr = vdd$k_begin;
            ch_ptr += sizeof (uint8_t);
            *(uint32_t*)ch_ptr = vdd$k_acl_dsc;
            ch_ptr += sizeof (uint32_t);
            *(uint32_t*)ch_ptr = 1;
            ch_ptr += sizeof (uint32_t);
            *(uint32_t*)ch_ptr = 0;
            ch_ptr += sizeof (uint32_t);
            {
                char acl_sys_all[] =
                    "(IDENTIFIER=[SYSTEM],ACCESS=R+W+E+D)";

                struct dsc$descriptor ace_str_dsc;
                ace_str_dsc.dsc$w_length  = sizeof (acl_sys_all) - 1;
                ace_str_dsc.dsc$b_class   = DSC$K_CLASS_S;
                ace_str_dsc.dsc$b_dtype   = DSC$K_DTYPE_T;
                ace_str_dsc.dsc$a_pointer = acl_sys_all;

                struct dsc$descriptor_s ace_bin_dsc;
                ace_bin_dsc.dsc$w_length  = 12;
                ace_bin_dsc.dsc$b_class   = DSC$K_CLASS_S;
                ace_bin_dsc.dsc$b_dtype   = DSC$K_DTYPE_T;
                ace_bin_dsc.dsc$a_pointer = (char*)ch_ptr;

                sys$parse_acl(&ace_str_d, &ace_bin_d,
                              NULL, NULL, NULL);
                ch_ptr += 12;
            }
            *(uint8_t*)ch_ptr = vdd$k_eoc;
            ch_ptr += sizeof (uint8_t);
        }
        *(uint8_t*)ch_ptr = vdd$k_end;
        ch_ptr += sizeof (uint8_t);
    }
    *(uint8_t*)ch_ptr = vdd$k_eoc;
    ch_ptr += sizeof (uint8_t);
}

unsigned short ch_diff;
ch_diff = ch_ptr - ch_buff;
```

```
struct dsc$descriptor ch_dsc;
ch_dsc.dsc$w_length  = ch_diff;
ch_dsc.dsc$b_class   = DSC$K_CLASS_S;
ch_dsc.dsc$b_dtype   = DSC$K_DTYPE_T;
ch_dsc.dsc$a_pointer = (char*)ch_buff;

long status;
status = vdd$change_element(message_vector, element_handle, &ch_dsc);
if (!(status & 1)) {
    lib$signal(status);
}
```

This example tries to change the ACL entry of an element. In the buffer we can see the use of the `acl_dsc` token, which shows the procedure that we want to change the ACL buffer there. The code tries to obtain the binary representation of the ACL string by calling the SYS$PARSE_ACL system service. Later on, it puts the parsed entry into the main VDD Buffer.

We can see that in the buffer, the `attribute_change` token is being glued together. This is because the VDD ACL Buffer is represented as an attribute inside of the VDD database. In fact, the buffer gets extracted from line 31 to line 60 and gets stored in the database without any modifications.

Upon receiving a change request from any VDD procedures, the system checks the list of specified ACEs via the SYS$CHKPRO system routine.

To apply the changes, we use the VDD$CHANGE_ELEMENT routine. The routine needs the `element_handle` of our element. This can be obtained via the VDD Fetch procedures.

5.7. VDD Element History

Each time an element gets created or modified, a history entry is attached to it. The entry includes the element ID, element type, created date, and modified date fields.

Chapter 6. Internal Architecture

This chapter will briefly discuss the internal architecture of VDD.

6.1. The VDD Context

Each initialization procedure of VDD creates a VDD context, which is a structure of various internal fields that get used during VDD operations. The reference to the *longword* variable that needs to be passed to VDD$SIGN_IN or VDD$ATTACH procedures is used as a pointer to that data structure. This data structure is called "VDD Context". It contains the following fields:

- The working directory path
- The list of tracked user data
- The list of "VDD Processes"
- The list of shared sessions
- Other minor book-keeping fields

The working directory path is used by various procedures when they need to deal with relative paths. It is also used by database routines that need to know where the database folder is. The default value for this field is the VDD$DEFAULT symbol.

The *VDD User Data* name is somewhat misleading because it is in reality used for legacy node identifiers. These are used each time a legacy procedure is used where it requires a node ID. The `node_id` argument is a pointer to the "VDD User Data" structure. The "VDD User Data" entity contains the following relevant fields:

- The magic number
- A pointer to the "VDD Context"
- A pointer to the "VDD Process"
- The type
- The flags
- Optional fields for more complex inherited structures

The magic number is a simple identifier generated internally for validation purposes. The two pointers are simple references to the parent data structures. This data type is used for the legacy APIs. It determines which VDD type of a node you have. Lastly, the flags field is used for locking and "tooling" mechanisms.

The list of sessions in "VDD Context" stores information about the current session. The most important fields are: `transaction bit`, `read-only bit`, `session id`, and `fetch handles`. A "VDD Fetch Handle" stores intermediate information for fetching routines.

6.2. The VDD Process

Each "VDD Context" has its own list of "VDD Processes". This data structure is allocated each time a "VDD Operation" is performed. The data structure, along with its state, gets destroyed after the operation completes. The structure consists of the following entities:

- Current DB structure
- A pointer to the parent "VDD Context"
- A message handler
- Various caches for fetch optimizations

The DB structure is used for database operations, which make sure that state stored in memory gets correctly saved to disk. The pointer to the context is a simple reference. The message handler is used for error/message logging purposes.

6.3. VDD Helper Modules

To assist with parsing and disk management, VDD implements a set of helper libraries. For the grammar parsing VDD uses the open source Bison parser, and for the database SQLite is used. To avoid feature gaps or library bugs, VDD modifies the source code to fit its needs. These modified versions of structures are usually wrapped into custom-made modules.

Chapter 7. Using VDD With Layered Products

The following sections detail using VDD with various layered products.

7.1. ACMS Applications With VDD

Important

VDD support for ACMS applications is currently only available for x86-64 systems.

The VSI Application Control Management System (ACMS) layered product provides multiple tools for creating applications. Definitions of these applications (services) are written in ACMS-specific syntax using the Application Definition Utility (ADU).

For more information about ACMS, see https://products.vmssoftware.com/acms or contact VSI.

Definitions, along with metadata about these services, are stored in the dictionary attached to ACMS at the point of execution (VDD in our case).

Warning

Replacing underlying dictionaries and libraries during runtime execution of any ACMS tool can and will lead to corrupted database and malformed application definitions.

7.1.1. Building ACMS Examples

There are five available examples of ACMS applications that users can try to build, provided that ACMS and VDD are installed on their system.

7.1.2. Prerequisites

Before you build the ACMS examples with VDD, make sure of the following:

- You have an up-to-date dictionary from the latest kit.
- You have started up the VDD system, defining the necessary logical names for VDD to function correctly. Use the following commands:

```
$ @SYS$STARTUP:VDD$STARTUP.COM
$ @SYS$MANAGER:VDD$SETUP_USER_ENV.COM
```

- You have run the ACMS runtime required records. Use the following commands:

```
$ VDO @ACMSELSTR.DDL
$ VDO @ACMPRSTAT.DDL
$ VDO @ACMTSKINF.DDL
```

- The CDDSHR logical name points to SYS$LIBRARY:VDDSHR.EXE.

7.1.3. ACMS Examples

Example 7.1. Sum of Two

The following example creates a simple ACMS application that takes two numbers as parameters and returns their sum:

```
$ VDO @ADD_DATA.VDO
DEFINE FIELD DATA DATATYPE IS LONGWORD.

$ VDO @ADD_NUMBER.VDO
DEFINE RECORD ADD_NUMBER.
    DATA.
END RECORD.

$ ADU @ADD_TASK.TDF
$ ADU @ADD_TASK_GROUP.GDF
%ADU-I-NONODWLCR, OBJECT 'ADD_TASK_GROUP' DOES NOT EXIST, CREATING OBJECT

$ ADU BUILD GROUP/STDL ADD_TASK_GROUP ADD_TASK_GROUP.TDB
%ACMSTDU-I-WRITETDB, WRITING TDB
-ACMSTDU-I-BYTESWRIT,   3584 BYTES (7 BLOCKS)
%ACMSTDU-S-WEB_WRITTEN, WDB FILE WRITTEN FOR STDL
%ACMSTDU-I-OBJMODCRE, PROCEDURE SERVER 'ADD_SERVER' OBJECT MODULE CREATED IN FILE
 EXAMPLE$ROOT:[EX1]ADD_SERVER.OBJ;5'

$ ADU @ADD_ACMS_APPL.ADF
%ADU-I-NONODWLCR, OBJECT 'VDD_TEST_APPL' DOES NOT EXIST, CREATING OBJECT

$ ADU BUILD APPLICATION/STDL VDD_TEST_APPL VDD_TEST_APPL.ADB
%ACMSCDU-I-WRITEADB, WRITING ADB
-ACMSCDU-I-BYTESWRIT,   1780 BYTES (4 BLOCKS)
%ACMSCDU-I-WRITESTDL, WRITING STDL
-ACMSCDU-I-BYTESWRIT,   1450 BYTES (3 BLOCKS)

$ CC ADD.C
$ LINK/SEGMENT_ATTRIBUTE=CODE=P0/EXE=ADD_SERVER.EXE ADD.OBJ,ADD_SERVER.OBJ
```

Important

The **/SEGMENT_ATTRIBUTE=CODE=P0** qualifier *must* be used together with the **LINK** command on x86-64 systems. Failure to include this qualifier will cause the ACMS application to crash.

Example 7.2. ACMS$IVP

The following example builds the IVP provided by the ACMS installation pack using VDD:

```
$ ADU @IVPGROUP1.COM
%ADU-I-NONODWLCR, OBJECT 'CDD$TOP.ACMS$DIR.ACMS$IVP.GROUP1' DOES NOT EXIST, CREATING
 OBJECT
%ADU-I-NONODWLCR, OBJECT 'CDD$TOP.ACMS$DIR.ACMS$IVP.TASK1' DOES NOT EXIST, CREATING
 OBJECT
$ ADU BUILD GROUP ACMS$DIR.ACMS$IVP.GROUP1 GROUP1/NODEBUG
%ACMSTDU-I-WRITETDB, WRITING TDB
-ACMSTDU-I-BYTESWRIT,   3584 BYTES (7 BLOCKS)
%ACMSTDU-I-OBJMODCRE, PROCEDURE SERVER 'GROUP1_SERVER1' OBJECT MODULE CREATED IN FILE
 EXAMPLE$ROOT:[ACMSREQ]GROUP1_SERVER1.OBJ;32'
$ ADU @IVPADB.COM
%ADU-I-NONODWLCR, OBJECT 'CDD$TOP.ACMS$DIR.ACMS$IVP.APPLICATION1' DOES NOT EXIST,
 CREATING OBJECT
$ ADU BUILD APPLICATION CDD$TOP.ACMS$DIR.ACMS$IVP.APPLICATION1 ACMSIVP
```

```
%ACMSCDU-I-WRITEADB, WRITING ADB
-ACMSCDU-I-BYTESWRIT,   1824 BYTES (4 BLOCKS)
```

7.1.4. Verifying ACMS Examples

Once an ACMS application is installed, you need to verify that the application functions correctly. To do so, run a test client application which connects to the ACMS TP gateway and then execute the ACMS application.

ACMS provides a special client library to allow for calling remote procedures with a specific interface.

The following example shows a client code written in C which calls a remote procedure from *Example 7.1, "Sum of Two"* using TP gateway:

```
#include <stdio.h>
#include <string.h>
#include <lib$routines.h>
#include "acmsdi.h"
#ifdef __VMS
#pragma member_alignment save
#pragma nomember_alignment
#endif
#pragma dictionary "add_number"
#ifdef __VMS
#pragma member_alignment restore
#endif

#define MAX_SEL_STRING 80

#define HOSTNAME "10.11.108.21"
#define USERNAME "TONYSPARK"
#define PASSWORD "VERYSECRETPASSWORD"

int main()
{
    char ss[MAX_SEL_STRING + 1] = "", sm[ACMSDI_STATUS_MSG_LEN + 1] = "";
    int rv;
    ACMSDI_SUBMITTER_ID sid;
    int i;
    int j;
    struct add_number arr[3];
    ACMSDI_WORKSPACE wksp[3];
    arr[0].data = 25;
    arr[1].data = 8;
    arr[2].data = 0;

    rv = acmsdi_sign_in(HOSTNAME, USERNAME, PASSWORD, 0, &sid, 0, NULL, NULL);
    fprintf(stderr, "[SIGN_IN]\t rv: %d\n", rv);
    if (rv != ACMSDI_NORMAL) {
        fprintf(stderr, "[SIGN_IN] Fatal error, cannot advace... Exiting.\n");
        return rv;
    }

    for (int i = 0; i < 3; i++) {
        ACMSDI_INIT_WORKSPACE(wksp[i], arr[i]);
    }

    rv = acmsdi_call_task (&sid, NULL, "ADD_TASK", "VDD_TEST_APPL", ss, sm, 3,
         &wksp, 0, NULL, NULL,NULL);

    fprintf(stderr, "[CALL_TASK]\t rv: %d\n", rv);
    for (j = 0; j < 3; j++) {
        printf("num%d = %lu\n", j, arr[j].data);
    }
```

```
    rv = acmsdi_sign_out(&sid, NULL, NULL, NULL);
    fprintf(stderr, "[SIGN_OUT]\t rv: %d\n", rv);

    return (0);
}
```

To compile and link this single C file, use the following commands:

```
$ CC/INCLUDE=ACMSDI$ROOT:[INCLUDE] DEMO01.C
$ LINK DEMO01.OBJ,SYS$INPUT/OPT ACMSDI$CLIENT_SHR/SHARE
```

Make sure that you have the ACMSDI gateway running. Otherwise, invoke the following procedure:

```
$ @SYS$STARTUP:ACMSDI$STARTUP.COM
```

Finally, run the demo application to test how VDD and ACMS work together:

```
$ RUN DEMO01
[SIGN_IN]         rv: 0
[CALL_TASK]       rv: 0
num0 = 25    ! Input Parameter 1
num1 = 8     ! Input Parameter 2
num2 = 33    ! This number was a result of RPC
[SIGN_OUT]        rv: 0
```

7.2. TDMS With VDD

Important

VDD support for TDMS is currently only available for x86-64 systems.

The VSI Terminal Data Management System (TDMS) layered product is designed for the implementation of interactive, forms intensive applications. It provides application programmers with a set of development tools to create and maintain forms based user interfaces, and VSI TDMS RT provides a runtime system for displaying and managing the user interface at execution time.

For more information about TDMS, see https://products.vmssoftware.com/tdms or contact VSI.

Warning

Currently, users cannot rebuild the TDMS examples in the TDMS$SAMPLES directory using VDD. These examples will be provided by the TDMS kit, already built, into the TDMS$SAMPLES directory. The user can then simply run them to test functionality.

7.3. Datatrieve With VDD

Important

VDD support for Datatrieve is currently only available for x86-64 systems.

The VSI Datatrieve layered product is a database query, report, and data management tool.

For more information about Datatrieve, see https://products.vmssoftware.com/datatrieve or contact VSI.

7.3.1. Prerequisites

Important

VDD X1.2-03 is required to use the latest Datatrieve T7.4-4 kit.

Before running the Datatrieve test with VDD, make sure of the following:

- You have started up the VDD system, defining the necessary logical names for VDD to function correctly. Use the following commands:

```
$ @SYS$STARTUP:VDD$STARTUP.COM
$ @SYS$MANAGER:VDD$SETUP_USER_ENV.COM
```

- The VDD$DEFAULT logical name points to the system repository (SYS$COMMON:[SYSTEMREPO]).

7.3.2. Setting Up Datatrieve

To set up the required environment for the Datatrieve test, run the NEWUSER.COM executable by entering the following command:

```
$ @DTR$LIBRARY:NEWUSER
```

You should see output similar to the following:

```
        NEWUSER creates the environment to help new users to get started
        with DATATRIEVE. It gives you the necessary files to perform
        the introductory examples in the DATATRIEVE documentation.

        This procedure will copy the DATATRIEVE data files from
        DTR$LIBRARY to your directory and will create the DATATRIEVE
        objects into your dictionary.

        You now have the following options:

        1.      Copy the DATATRIEVE data files to your directory and
                create the DATATRIEVE objects into your dictionary.

        2.      Copy only the DATATRIEVE data files to your directory.

        3.      Create only the DATATRIEVE objects into your dictionary.

        E.      None of the above and EXIT.

        Select option: 1

    Your default directory is [USER.EXAMPLE]

        Do you want to copy the DATATRIEVE data files there ? (Y/N) Y

    Your default Dictionary directory is SYS$COMMON:[SYSTEMREPO]

        Do you want to define the DATATRIEVE objects there ? (Y/N) Y
```

```
    NEWUSER is working... It will take a few minutes.


VSI DATATRIEVE T7.4-3
Digital Query and Report System
Type HELP for help

[Record is 104 bytes long.]
[Record is 382 bytes long.]
[Record is 56 bytes long.]
[Record is 57 bytes long.]
[Record is 18 bytes long.]
[Record is 35 bytes long.]
[Record is 624 bytes long.]
[Record is 41 bytes long.]
[Record is 47 bytes long.]
[Record is 142 bytes long.]



        The following command has been defined for you, but you will need
        to add it to your LOGIN.COM file for the next time you log in:


        DEFINE/PROCESS VDD$DEFAULT SYS$COMMON:[SYSTEMREPO]


    All data copied successfully.

    If you need help, see the person responsible for DATATRIEVE on your system.

    To invoke DATATRIEVE just type:     DATATRIEVE
```

7.3.3. Testing Datatrieve

Complete the following test to ensure that Datatrieve is working correctly:

```
$ DATATRIEVE
VSI DATATRIEVE T7.4-3
Digital Query and Report System
Type HELP for help
DTR> PRINT "TODAY" USING DD-MMM-YYYYBBW(9)
5-Oct-2025  Sunday

DTR> SHOW YACHT
RECORD YACHT USING
01 BOAT.
  03 TYPE.
     06 MANUFACTURER PIC X(10)
        QUERY_NAME IS BUILDER.
     06 MODEL PIC X(10).
  03 SPECIFICATIONS
     QUERY_NAME SPECS.
     06 RIG PIC X(6)
        VALID IF RIG CONT "SLOOP","KETCH","MS","YAWL".
     06 LENGTH_OVER_ALL PIC XXX
        VALID IF LOA BETWEEN 15 AND 50
        QUERY_NAME IS LOA.
     06 DISPLACEMENT PIC 99999
        QUERY_HEADER IS "WEIGHT"
```

```
        EDIT_STRING IS ZZ,ZZ9
        QUERY_NAME IS DISP.
     06 BEAM PIC 99 MISSING VALUE IS 0.
     06 PRICE PIC 99999
        MISSING VALUE IS 0
        VALID IF PRICE>DISP*1.3 OR PRICE EQ 0
        EDIT_STRING IS $$$,$$$.
;
DTR> READY YACHTS
DTR> PRINT AVERAGE PRICE OF YACHTS

AVERAGE
PRICE

[Function computed using 50 of 113 values.]
$25,389

DTR>
```

Chapter 8. Building VMS Applications

To build a native compiler application on OpenVMS, you must write and run a VDO script that creates records for that compiler in the VSI Data Dictionary. Sections below go over that process in detail.

To run a VDO script, enter the following command:

```
$ VDO @script-file-name.VDO
```

8.1. Building a BASIC Application

Running the VDO script below will add records for the BASIC compiler to VDD and make it possible to run BASIC applications:

```
DEFINE RECORD ACCOUNT_LOCATION_BASIC.
       ACCOUNT_NUMBER  DATATYPE IS LONGWORD.
   BILLING_ADDRESS STRUCTURE.
                ADDRESS_LINE    DATATYPE IS TEXT
                        SIZE IS 40 CHARACTERS OCCURS 4 TIMES.
                REGION          DATATYPE IS TEXT
                        SIZE IS 30 CHARACTERS.
                POST_CODE       DATATYPE IS WORD.
        END BILLING_ADDRESS STRUCTURE.
        DEPOT_ADDRESS STRUCTURE OCCURS 10 TIMES.
                ADDRESS_LINE    DATATYPE IS TEXT
                        SIZE IS 40 CHARACTERS OCCURS 4 TIMES.
        REGION          DATATYPE IS TEXT
                        SIZE IS 30 CHARACTERS.
        POST_CODE       DATATYPE IS WORD.
        END DEPOT_ADDRESS STRUCTURE.
        APPLEWORD       DATATYPE IS SIGNED WORD.
        LEMONLONG       DATATYPE IS SIGNED LONGWORD.
        QUINCEQUAD      DATATYPE IS SIGNED QUADWORD.
        BLUEBERRYBYTE   DATATYPE IS SIGNED BYTE.
        OVERDUE         DATATYPE IS PACKED DECIMAL SIZE IS 10 DIGITS SCALE -2.
        COMMENT         DATATYPE IS TEXT
                        SIZE IS 200 CHARACTERS.
        DELTA           DATATYPE IS D_FLOATING.
        FIBONACCI       DATATYPE IS F_FLOATING.
        GALILEO         DATATYPE IS G_FLOATING.
END ACCOUNT_LOCATION_BASIC RECORD.
```

After running the above VDO script, VSI recommends that you create and run a simple "hello world" BASIC application to ensure that the compiler works correctly with VDD. For example, save the following script as the file BASIC-TEST.BAS:

```
program test
option type = explicit
%include %from %cdd "account_location_basic"
print "Hello world"
end program
```

and then run it via the following command:

```
$ BASIC/LIST/SHOW=CDD_DEFINITIONS BASIC-TEST.BAS
```

8.2. Building a FORTRAN Application

Running the VDO script below will add records for the FORTRAN compiler to VDD and make it possible to run FORTRAN applications:

```
DEFINE RECORD OCC_for_USER_EXTRACT.
    OCC_for_USER_EXTRACT STRUCTURE.
        SORT_IND    DATATYPE IS signed byte.
        OCC_USER_ID DATATYPE IS TEXT
                    SIZE IS 12 CHARACTERS.
        BILL_CYCLE  DATATYPE IS TEXT
                    SIZE IS 2 CHARACTERS.
        CUST_ACCT   DATATYPE IS signed longword.
        ACCT_STATUS DATATYPE IS signed word.
        ACCT_NAME   DATATYPE IS TEXT
                    SIZE IS 20 CHARACTERS.
        LAST_UPDATE_DATE DATATYPE IS text
                    SIZE IS 8 CHARACTERS.
        LAST_UPDATE_TIME DATATYPE IS text
                    SIZE IS 5 CHARACTERS.
        OCC_DATE STRUCTURE.
            OCC_CC      DATATYPE IS signed word.
            OCC_YR      DATATYPE IS signed word.
            OCC_MO      DATATYPE IS signed word.
            OCC_DAY     DATATYPE IS signed word.
        END OCC_DATE STRUCTURE.
        OCC_AMOUNT  DATATYPE IS f_floating.
        OCC_STATUS  DATATYPE IS text
                    SIZE IS 2 CHARACTERS.
        OCC_CODE    DATATYPE IS TEXT
                    SIZE IS 3 CHARACTERS.
        OCC_COMMENTS DATATYPE IS text
                    SIZE IS 20 CHARACTERS.
    END OCC_for_USER_EXTRACT STRUCTURE.
END OCC_for_USER_EXTRACT RECORD.
```

After running the above VDO script, VSI recommends that you create and run a simple "hello world" FORTRAN application to ensure that the compiler works correctly with VDD. For example, save the following script as the file FORTRAN-TEST.FOR:

```
options /extend_source
program test
implicit none
dictionary "occ_for_user_extract"
print *, 'Hello world!'
end
```

and then run it via the following command:

```
$ FORTRAN/LIST/SHOW=DICTIONARY FORTRAN-TEST.FOR
```