

# Chapter 36: Table Files

This chapter describes table files. Table files contain product-specific, installation-independent information. Most, but not all, products require a table file. **UPS** product developers are responsible for providing the table files associated with their products.

## 36.1 About Table Files

---

Table files are created and maintained by product developers. Table files contain the non-system-specific and non-shell-specific information that **UPS** uses for installing, initializing, and otherwise operating on product instances. For a given product, usually a single table file suffices for several instances, especially of a single version. Sometimes each instance has a separate table file. Table file names are arbitrary; we present recommendations in section 36.3 *Recommendations for Creating Table Files*.

Typically, when a **UPS** command is issued, **UPS** finds the table location from the command line or the version file (see section 29.4 *Determination of ups Directory and Table File Locations*). The command completes its internal processes, and then within the table file, it proceeds to:

- 1) locate the stanza that matches the specified product instance
- 2) find an **ACTION** keyword value that corresponds to the command, if any (see Chapter 34: *Actions and ACTION Keyword Values*)
- 3) execute the functions listed underneath the corresponding **ACTION** keyword, if any (see Chapter 35: *Functions used in Actions*), or
- 4) reverse the functions listed underneath the **ACTION** corresponding to the “uncommand” (see section 34.2.2 *“Uncommands” as Actions*)

## 36.2 When Do You Need to Provide a Table File?

---

Not all products require a table file. In particular, if *no* processing besides the internals and defaults needs to be done for *any* **UPS** command run on a particular product, and if its `ups` directory and documentation reside in the default areas, then the product doesn't need a table file. However, for products that do need a table file (most), at least a rudimentary table file must be in place before any instance is declared to a target **UPS** database. If it's not added right away, users may see incorrect behavior before it is there.

## 36.3 Recommendations for Creating Table Files

---

- Although table files can have any file name, we recommend that they be named as `<product>.table` (e.g., `emacs.table`) or `<version>.table` (e.g., `v19_34b.table`) for easy identification. If a table file is unique to a particular version of the product (which is likely because versions of product dependencies often change along with the version of the main product) then the name should be `<product>_<version>.table` (e.g., `emacs_v19_34b.table`).
- Table files should not source any `setup.[c]sh` script unless flow control (if then else, looping, etc.) is needed. For assistance, contact [uas-group@fnal.gov](mailto:uas-group@fnal.gov).
- In most cases, “un” actions (e.g., `UNSETUP`, `UNCURRENT`) are not needed (see section 34.2.2 “*Uncommands*” as Actions). If an “un” action is *not* specified in the table file, **UPS** will undo what the corresponding action did (e.g., `SETUP`, `CURRENT`), in reverse order, provided reversible functions were used (see section 35.2 *Reversible Functions*).
- Individual groups or experiments at Fermilab may set standards regarding table files that members should follow; contact your group leader to find out if there are any you need to be aware of. For example, ODS prefers that table files be maintained in the **UPS** database product subdirectory (e.g., `$PRODUCTS/emacs`) rather than in the product's `ups` directory.

## 36.4 Table File Structure and Contents

---

### 36.4.1 Basic Structure

The file starts with a header that identifies the file type and the product:

```
File=Table
Product=<product>
```

The basic structure of table file contents consists of an instance identifier followed by one or more actions (described in Chapter 34: *Actions and ACTION Keyword Values*). By the time **UPS** accesses the table file, it has already determined the database, product name and product version. Therefore **FLAVOR** and **QUALIFIERS** together are sufficient to identify the instance.

Here is a sample table file that illustrates the basic structure:

```
File=Table
Product=exmh

FLAVOR=SunOS+5
QUALIFIERS=" "

ACTION=SETUP
    setupRequired(expect)
    setupRequired(mh)
    ...
ACTION=UNSETUP
    ...
```

User-defined keywords, described in section 28.2 *Keywords: Information Storage Format*, can also be included after an instance identifier for use within actions.

## 36.4.2 Grouping Information

When a single table file represents multiple instances, a grouping structure can be superimposed on this basic structure to organize the information. To avoid having to repeat identical actions for a series of FLAVOR/QUALIFIER identifiers, the keyword FLAVOR can take the value ANY in table files. FLAVOR=ANY is taken as a best match, assuming all other instance identifiers match (see Chapter 27: *Product Instance Matching in UPS/UPD Commands* for more information on instance selection).

Grouping information within table files is supported via the use of the following three markers:

- GROUP:** Groups together multiple flavor/qualifier pairs. All entries subsequent to GROUP: are part of this group until an END: marker is found.
- COMMON:** Groups together actions that apply to all instances represented in GROUP:. COMMON: is only valid within a GROUP:.
- END:** Marks the end of a GROUP: or COMMON:. One END: marker is used to jointly end a GROUP: and an included COMMON:.

**UPS** does not *require* grouping in table files; these markers are available for convenience and for organizing information clearly. However, if GROUP: or COMMON: is used, END: must appear at the end of it, even if that is the very end of the file.

### 36.4.3 The Order of Elements

Blank lines are ignored, and therefore can be placed anywhere.

- The first keywords after GROUP: must always be FLAVOR followed by QUALIFIERS (i.e., the instance identifiers).
- FLAVOR and QUALIFIERS *cannot* be included within a COMMON: grouping.
- User-defined keywords can be defined anywhere except between GROUP: and the instance identifiers.
- Actions (described in Chapter 34: *Actions and ACTION Keyword Values*) for each instance are located after the instance-identifying keywords, and often between a COMMON: and END:.
- All actions after COMMON: apply to all the FLAVOR-QUALIFIERS pairs listed above it within the current GROUP:.
- All statements apply to the most recently defined FLAVOR/QUALIFIER keywords except for the statements between COMMON: and END: (which apply to all the flavors in the current GROUP:)
- GROUP:s cannot be nested.

## 36.5 Product Dependencies

---

### 36.5.1 Defining Dependencies

UPS product dependencies get listed in the SETUP action for the product instance in question. The `setupRequired` and `setupOptional` functions, described in section 35.3 *Function Descriptions*, can be used within the SETUP action to setup the dependencies along with the main product. These two functions take the same set of options and arguments as a normal `setup` command (see section 23.1 *setup*) in order to clearly specify the desired instance of the dependent product. We discourage specification of particular versions of products, and recommend using chains instead, e.g.,:

```
ACTION=SETUP
    setupRequired("perl")
```

This example sets up the default instance of `perl`, chained to current. Using chains, it is easier to keep the dependencies and the main product in sync.

Products that are not maintained in the UPS framework can also be designated as dependencies. You would need to use the function `exeAccess` to locate and access a non-UPS executable through your \$PATH. For example, the action:

```
ACTION=SETUP
  setupOptional(gcc)
  exeAccess(gcc)
```

tells **UPS** to setup the current instance of **gcc** if there is one declared; the **exeAccess** function checks for a version of **gcc** in your **\$PATH**, even if it's not one that is managed by **UPS**, and exits with an error if one is not found.

## 36.5.2 Product Dependency Conflicts

When different dependencies include the same product via different dependency trees (and therefore may require different instances of the same product), rules have been established to determine which instance of the dependent product is selected and in which order the required products are setup.

## Selection Algorithm for Conflicting Dependencies

The rules are as follows:

- 1) First level product dependencies, defined as those products listed as dependencies in the table file of the main product instance, take precedence over lower level dependencies when selecting which instance of the required product to set up.
- 2) Dependencies listed later in the table file take precedence over those listed earlier.

### Example of Dependency Selection and Order of Setup

We'll take you through an example that illustrates how the dependencies are selected and in what order they are setup. Our sample dependency structure starts with the product **A** as the parent product. It has two dependencies, which in turn have dependencies of their own. **B b1** refers to product **B**, version **b1**, and so on. (We recommend that developers avoid using specific version dependencies in general; we use them in our example for illustrative purposes.) Some of the dependencies are conflicting:

In **A**'s table file:

```
product A
setupRequired(B b1)
setupRequired(C c1)
```

In **B b1**'s table file:

```
product B b1
setupRequired(C c2)
setupRequired(D d1)
```

In **C c2**'s table file:

```
product C c2
setupRequired(D d3)
```

In **C c1**'s table file:

```
product C c1
setupRequired(D d2)
```

The tree is traversed starting at **A**, then going down each dependency branch. So the order in which the products are encountered is:

- 1) **A** (no conflict)
- 2) **A**'s dependencies **B b1** and **C c1** are selected since they are the highest level dependencies.
- 3) Start down **B b1** branch: find **C c2** (version **c1** already selected by rule 1; **C c2** ignored)

- 4) Completing the **B b1** branch, find **D d1**. It is examined, and ultimately passed over (by rule 2) because **D d2**, a dependency of **C c1** and therefore also a second-level dependency of **A**, is encountered later.

## 36.6 Keywords that can be Used in Table Files

---

Keyword and Default Value (if any)	Description and Notes (if any)
ACTION	defines an action (described in Chapter 34: <i>Actions and ACTION Keyword Values</i> ), i.e., groups together a list of functions associated with a command (e.g., ACTION=SETUP)
CATMAN_SOURCE_DIR Default: under the \${UPS_UPS_DIR}/ toman/catman directory	location of catman files (formatted man page files) included with instance
COMMON:	groups together actions that apply to all instances represented in "GROUP:"; COMMON: is only valid within a GROUP:
DESCRIPTION	product description
END:	marks the end of a "GROUP:" or "COMMON:"; one "END:" marker is used to jointly end a "GROUP:" and an included "COMMON:"
FILE	type of file (possible values: DBCONFIG, UPDCONFIG, CHAIN, VERSION, TABLE)
FLAVOR	product instance flavor Note: To easily accommodate flavor-neutral <b>setup</b> functions in a table file, FLAVOR can take the value ANY, but <i>only</i> in a table file.
GROUP:	groups together multiple instances; all entries subsequent to this "GROUP:" are part of it until an "END:" marker is reached
HTML_SOURCE_DIR Default: under the \${UPS_UPS_DIR}/ tohtml directory	location of html files included with instance <i>not supported in UPS v4</i>

Keyword and Default Value (if any)	Description and Notes (if any)
INFO_SOURCE_DIR Default: under the \${UPS_UPS_DIR} / toInfo directory	location of Info files included with instance
MAN_SOURCE_DIR Default: under the \${UPS_UPS_DIR} / toman/man directory	location of unformatted man page files included with instance
NEWS_SOURCE_DIR Default: under the \${UPS_UPS_DIR} / tonews directory	location of news files included with instance <i>not supported in UPS v4</i>
PRODUCT	product name
QUALIFIERS	additional instance specification information often used to indicate compilation options used by developer Notes: appears immediately after a FLAVOR in these files, and is coupled with it to complete the instance identification (see 27.2.3 <i>Qualifiers: Use in Instance Matching</i> )
UPS_DB_VERSION	UPS database version
USER	current username
VERSION	product version
_UPD_OVERLAY	main product name for overlaid product Note: This keyword is user-defined from UPS's point of view. It is included here because it is configured and used by UPD. Its use with overlaid products is described in section 28.6.6 <i>_UPD_OVERLAY</i> .

## 36.7 Table File Examples

---

### 36.7.1 Example Illustrating Use of FLAVOR=ANY

Below is a sample table file for the product **exmh** version v1\_6\_6 which uses FLAVOR=ANY. For the **exmh** instances whose version files point to this table file, all except those with qualifiers share the same stanza:

```
File=Table
Product=exmh
#*****
# Starting Group definition
```

```

Group:
Flavor=ANY
Qualifiers=" "

Common:
  Action=setup
    setupRequired(expect)
    setupRequired(mh)
    setupOptional(glimpse)
    setupOptional(www)
    setupOptional(mimertools)
    setupOptional(ispell)
    setupOptional(popclient)
    prodDir()
    setupEnv()
    pathPrepend(PATH, ${UPS_PROD_DIR}/bin)
  Action=configure
    execute(${UPS_PROD_DIR}/ups/configure, UPS_ENV)
End:

```

Actions, functions and variables as used in this example are described in Chapter 34: *Actions and ACTION Keyword Values*, section 35.3 *Function Descriptions* and section 35.6 *Local Read-Only Variables Available to Functions*, respectively.

You'll notice that there are no functions specified for **unsetup** in this table file. Due to the defaults that **UPS** has in place, when **unsetup** is run all of the **setup** functions will be reversed (the required products will get unsetup, the defined environment variables will get undefined, and the product's `bin` directory will be dropped from `$PATH`). See sections 34.2.2 "*Uncommands*" as *Actions* and 35.2 *Reversible Functions*.

## 36.7.2 Example Showing Grouping

Grouping is illustrated in the following example:

```

FILE=Table
PRODUCT=exmh
#*****
# Starting Group definition
GROUP:
FLAVOR=IRIX+5
QUALIFIERS=" "

FLAVOR=IRIX+5
QUALIFIERS="mips2"

```

```

COMMON:
    ACTION=SETUP
        setupOptional(expect)
        ...
    ACTION=CONFIGURE
        execute( ${UPS_PROD_DIR}/ups/configure,UPS_ENV)
        ...
END:
#*****
# Starting Group definition
GROUP:
FLAVOR=ANY
QUALIFIERS=" "

COMMON:
    ACTION=SETUP
        setupRequired(expect)
        ...
    ACTION=CONFIGURE
        execute( ${UPS_PROD_DIR}/ups/configure,UPS_ENV)
        ...
END:

```

The second group (defined by `FLAVOR=ANY`) matches all the instances not matched in the first group, except those with qualifiers.

### 36.7.3 Example with User-Defined Keywords

User-defined keywords are described in section 28.2 *Keywords: Information Storage Format*. All user-defined keywords must have underscore (`_`) as the initial character (e.g., `_dest_arch`). The following example illustrates their use in a table file:

```

File=Table
Product=vxboot
#*****
# Starting Group definition
Group:
Flavor=NULL
Qualifiers="narrow29"
    _dest_arch=ppc
    _dest_env=VxWorks-5.3
    _dest_type=MVME2301
...
Common:
    Action=setup
        setupEnv( )

```

```
envSet (VXB_DEST_ARCH, ${_dest_arch})  
envSet (VXB_DEST_ENV, ${_dest_env})  
envSet (VXB_DEST_TYPE, ${_dest_type})
```

...

## 36.7.4 Examples Illustrating ExeActionOpt Function

### Example 1

In this example, there are actions for the first two instance identifiers, but not for the third. We want to execute the XYZ action at setup time if it's there, but continue processing if it's not. To do this, we must call the action using the **exeActionOpt** function.

```
FILE=Table
PRODUCT=fred
#*****
# Starting Group definition
GROUP:
FLAVOR=SunOS+6
QUALIFIERS=" "
    ACTION=XYZ
        fileTest(/, -w, "You must be root to run this
command.")

FLAVOR=IRIX+6
QUALIFIERS=" "
    ACTION=XYZ
        fileTest(/, -w, "You must be root to run this
command.")

FLAVOR=IRIX+6
QUALIFIERS="mips2"
# No XYZ action

COMMON:
    ACTION=SETUP
        exeActionOpt(XYZ)

END:
...
```

### Example 2

In this example, we use the **exeActionOpt** function to instruct **UPS** to execute one action or another, depending on whether the user supplies an option on the **setup** command line.

```
FILE=Table
PRODUCT=fred
#*****
# Starting Group definition
GROUP:
```

```
FLAVOR=ANY
QUALIFIERS=" "

ACTION=SETUP
    exeActionOpt(XYZ_${UPS_OPTIONS})

ACTION=XYZ_
    function_1()
ACTION=XYZ_FULL_LICENSE
    function_2()
...
```

If you run:

```
% setup fred
```

you'll execute ACTION XYZ\_. To execute ACTION XYZ\_FULL\_LICENSE, you need to run:

```
% setup fred -O FULL_LICENSE
```

