

Chapter 34: Actions and ACTION Keyword

Values

Table files and **UPD** configuration files often include stanzas which we call *actions*. We describe actions in this chapter.

34.1 Overview of Actions

An action is a construction that identifies a **UPS** or user-defined operation via the **ACTION** keyword (defined in section 28.4 *List of Supported Keywords*), and lists functions to perform, in addition to any internal processes, when the operation is executed. An action can be called by a **UPS** command, a user-defined **UPS**-style command, or by another action. An action stanza has the format:

```
ACTION=<VALUE>
    <function_1>([<argument_1>] [, <argument_2>] ...)
    <function_2>([<argument_1>] [, <argument_2>] ...)
    ...
```

As for all keyword values, the **VALUE** is not case-sensitive. Nor are the functions, although some arguments are. The supported **ACTION** keyword values include:

- strings that correspond to **UPS** commands
- chains and “unchains” (explained in section 34.3.2 “*Unchains*” as *Actions*)
- user-defined strings handled by the *Unknown Command Handler*

The supported functions are listed in section 35.3 *Function Descriptions*.

34.2 UPS Command Actions

34.2.1 UPS Commands as Actions

Most commonly, the ACTION keyword value is a string that corresponds to a **UPS** command. The string is usually the command itself (minus the **ups** at the front, if it is part of the command), e.g., **SETUP**, **CONFIGURE**, **DECLARE**. The supported strings in this category include:

CONFIGURE and **UNCONFIGURE**

COPY

DECLARE and **UNDECLARE**

GET

MODIFY

SETUP and **UNSETUP**

START

STOP

TAILOR

The **UPS** commands that cannot have a corresponding action in a table file are: **ups flavor** and **ups help** (because no table file can be associated with them); **ups depend**, **ups list**, and **ups verify** (because they can operate on more than one database); and **ups exist**, **ups modify** and **ups touch**.

34.2.2 “Uncommands” as Actions

Several of the **UPS** commands have “uncommand” counterparts, namely **setup/unsetup**, **ups declare/undeclare**, **ups configure/unconfigure**. Generally, if the “uncommand” is expected to undo everything that the original command did, and only that, then including an ACTION=<UNCOMMAND> action in the table file is unnecessary.

Uncommands and Reversible Functions

If an “unaction” is not present, **UPS** will look for the corresponding ACTION=<COMMAND>, and undo all the reversible functions that were performed. In section 35.2 *Reversible Functions* we discuss reversible functions. If the “uncommand” needs to do something other than the exact reversal of the command, include an “unaction” for it (i.e., ACTION=<UNCOMMAND>) and specify the functions to execute.



This works both ways. Say the original command is “uncommand” (e.g., **ups undeclare**), and you have included `ACTION=<UNCOMMAND>` but not `ACTION=<COMMAND>` in the table file. Then when you run “command”, **UPS** will attempt to reverse all the functions listed under `ACTION=<UNCOMMAND>`.

Uncommands and Script Execution

For the functions **sourceOptCheck**, **sourceOptional**, **sourceReqCheck**, and **sourceRequired**, the “uncommand” will execute an “unscript” in a similar way. You do not have to specify an “unaction” in the table file as long as the scripts to source are in the same directory and have matching script and “unscript” filenames (i.e., `<scriptname>` and `un<scriptname>`). This also works both ways, as discussed above.

Here is an example. Say a `CONFIGURE` action specifies:

```
ACTION=CONFIGURE
```

```
sourceOptional(${UPS_UPS_DIR}/configure.${UPS_SHELL},UPS_ENV
)
```

When you run the **ups unconfigure** command, **UPS** first looks for `ACTION=UNCONFIGURE`, as usual. Failing to find it, **UPS** next looks for `ACTION=CONFIGURE`. Upon encountering the **sourceOptional** function and seeing that it sources the `configure.${UPS_SHELL}` script, **UPS** searches for the file `unconfigure.${UPS_SHELL}` in the same directory (`${UPS_UPS_DIR}`), and sources it.

34.3 Chain Actions

34.3.1 Chains as Keyword Values

Chain names are allowable as `ACTION` keyword values. This includes any predefined chain name (as listed in section 2.3.5 *Chains*: `CURRENT`, `TEST`, `DEVELOPMENT`, `OLD`, `NEW`) or any user-defined chain name (e.g., `MY_CHAIN`). Chain actions are executed when a chain of the corresponding name is declared to a product instance via the **ups declare** command. For example, if you declare an instance as `current`, **ups declare -c** looks for `ACTION=CURRENT`.



Sometimes a **UPS** command executes more than one action. For example, the **ups declare -c** command executes both the `CURRENT` and `DECLARE` actions, if they are present.

34.3.2 “Unchains” as Actions

Similarly, when a chain is removed from an instance (which can happen with either **ups declare** or **ups undeclare**), **UPS** looks for the corresponding chain name preceded by the “UN” prefix (e.g., UNCURRENT, UNTEST, UNMY_CHAIN).

The relationship between a chain action and its corresponding “unchain” action (e.g., CURRENT and UNCURRENT) is the same as between commands and “uncommands”, as described in section 34.2.2 “*Uncommands as Actions*”. For example, if an “unchain” action is sought but not found, **UPS** will then look for the corresponding ACTION=<CHAIN> and undo all the reversible functions listed there.

34.4 The “Unknown Command” Handler

The unknown command handler effectively allows you to define a **UPS**-like “unknown” command for use with a product. To define one, include in the product’s table file an ACTION with a unique value of your choosing, e.g., ACTION=XYZ. The corresponding command will be **ups xyz**. The action should contain one or more supported functions (listed in section 35.3 *Function Descriptions*), as usual. Here is an example of what the action may look like:

```
ACTION=XYZ
  envSet(VARIABLE, value)
  sourceRequired(SCRIP.T.csh, UPS_ENV_FLAG)
```

The command **ups xyz** is now available for you to use. Enough information must of course be provided on the command line to locate the table file containing the action, e.g.,:

```
% ups xyz [<options>] <product> [<version>]
```

When it is executed, the unknown command handler locates ACTION=XYZ in the table file and executes the functions listed under it.

User-defined ACTION keyword values (e.g., XYZ) do not need to start with underscore (_), as contrasted with user-defined *keywords* (see section 28.2 *Keywords: Information Storage Format*).

Examples

An example of the use of the unknown command handler can be found in the table file for the product **xemacs v20_4**:

```
ACTION=CONFIGURE
```

```

        Execute(echo "Do a 'ups blessmail xemacs' as root to
make mail work.",NO_UPS_ENV)
        ACTION=BLESSMAIL
        Execute(chgrp mail ${UPS_PROD_DIR}/lib/**/movemail,
NO_UPS_ENV)
        Execute(chmod 2755 ${UPS_PROD_DIR}/lib/**/movemail,
NO_UPS_ENV)

```

When the product instance is configured (via the first **ups declare**, or manually via the **ups configure** command), an **echo** command prints to screen an instruction to run the user-defined (“unknown”) command **ups blessmail**. This command is handled by the unknown command handler. It finds ACTION=BLESSMAIL and executes the functions associated with it. UPD’s table file includes (at least) two actions using the unknown command handler:

```

        action = installasroot
                Execute(${UPS_UPS_DIR}/setupautoupp localnode,
UPS_ENV)

        action = installprodserver
                Execute(${UPS_UPS_DIR}/setupautoupp productnode,
UPS_ENV)

```

If you’re installing on a local node, you’d run **ups installasroot upd** after installing UPD; if you’re installing it on a server node, you’d run **ups installprodserver upd** instead.

34.5 Actions Called by Other Actions

As mentioned in section 34.1 *Overview of Actions*, one action can execute another in the same file. The called action must be assigned a unique value of your choosing, e.g., ACTION=XYZ, and the calling action (or actions) must include one of the following functions (shown for ACTION=XYZ):

```
exeActionRequired("xyz")
```

or

```
exeActionOptional("xyz")
```

These functions are described in sections 35.3.14 *exeActionRequired* and 35.3.13 *exeActionOptional*, respectively.

This technique is useful in cases where two different **UPS** operations require overlapping functionality. For example, you may want one or more identical functions to be performed when a product gets configured and when it gets declared as current. The following example shows how to arrange this:

```
        action = configure
```

```
    <functions for configure>
    exeActionRequired("common")
action = current
    <functions for current>
    exeActionRequired("common")
action = common
    <functions common to both configure and current>
```