

# Creating and Configuring File System Dynamically Loadable Kernel Modules



## Table of contents

Abstract .....	2
Introduction .....	2
Creating a File System Loadable Module.....	3
Configuring a File System Loadable Module .....	7
Planning File System Layout on Machines with Loadable File System Modules.....	8
Summary .....	8
For more information .....	8

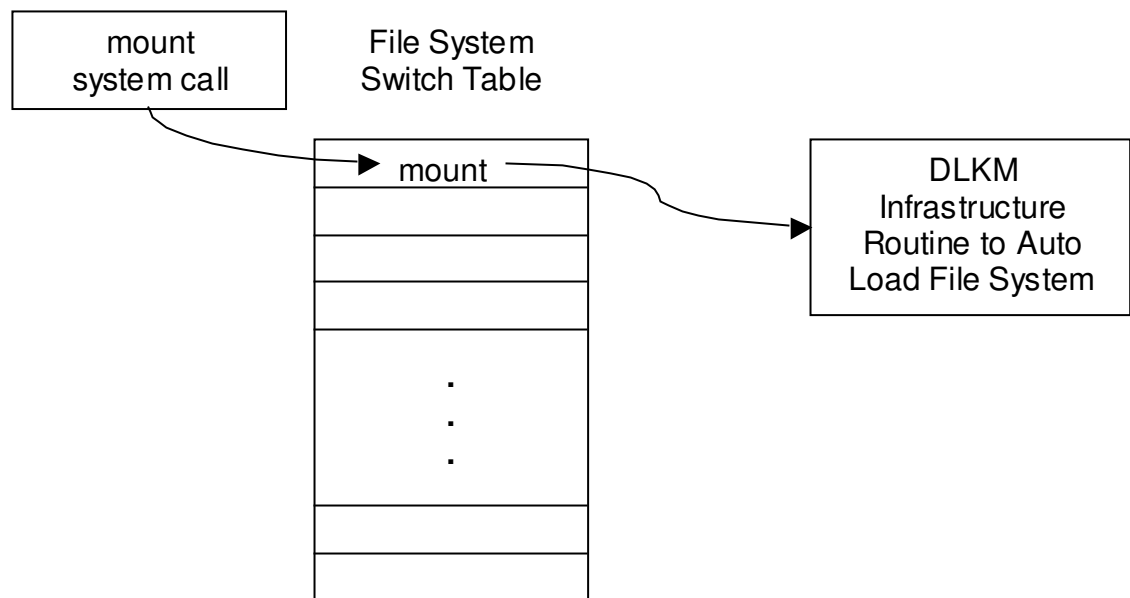
## Abstract

Users have long wanted systems that are robust and can operate with a minimum of down time. In an effort to help realize that goal dynamically loadable kernel modules (DLKM) were introduced to HP-UX in release 11.00. Initially, device driver and pseudo driver modules were supported. With the advent of HP-UX 11i v2, support for file system dynamically loadable kernel modules will be available. This white paper examines that capability and provides information on creating and configuring systems with dynamically loadable file systems such that a costly reboot can be avoided when file system kernel code needs to be updated.

## Introduction

The dynamically loadable kernel module (DLKM) feature was introduced to HP-UX in release 11.00. At its introduction only device driver and pseudo driver modules were supported. Drivers could be written in such a way that they would be brought into the running kernel only when needed. This “loading” could be done by an administrator or automatically, depending on configuration. While in use the driver remained loaded in the running kernel. When the driver was no longer needed it could be “unloaded” or removed from the kernel freeing valuable memory space. This also had the side effect of being able to replace certain modules without needing to reboot the machine. One could simply replace the object code of the DLKM and reload with a newer version.

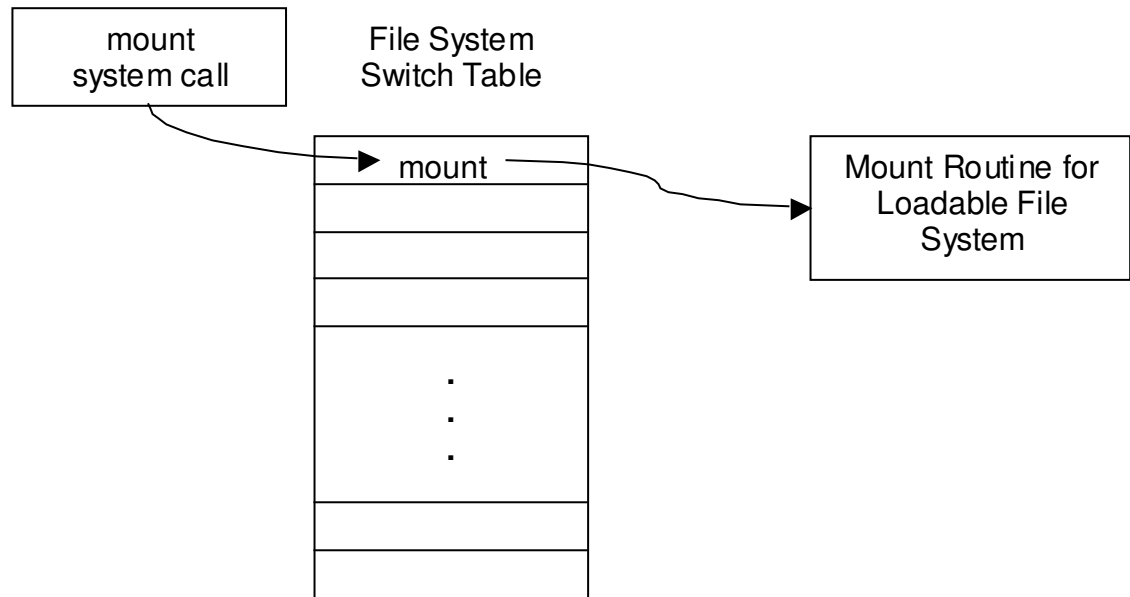
File systems are excellent candidates for this technology as by design. They are, by design, very modular and use a virtual file system (VFS) infrastructure. They also interface to the kernel through a switch table. Commands (e.g., mount, umount, etc.) that perform operations on the file system access the file system code using system calls that then call functions in the file system switch table. Certain functions in this table can be replaced by functions in the DLKM infrastructure that would force modules to be loaded when they are needed. Figure 1 shows the file system configuration prior to loading the actual file system module. The internal file system switch table contains a pointer to the routine in the kernel that forces the module to be loaded when the user invokes the mount system call.



**Figure 1: File system switch table configuration before loading**

This DLKM routine is responsible for getting the module into the running kernel and calling the file system load routine to get the switch table filled with pointers to the real file system code. Figure 2 shows the file switch table after the load has completed. The module’s load routine has installed the proper operations pointers in the file system switch

table. The file system is then mounted and normal operations can take place. Internally a count is kept of any references to the file system so that it cannot be unloaded until that count is zero. This is an important point to remember when planning your file system's layout because the module cannot be replaced until the file system is unmounted. When the module is unloaded, the mount routine pointer in the switch table is again replaced with the pointer to the routine to auto-load the file system.



**Figure 2: File system switch table after load complete**

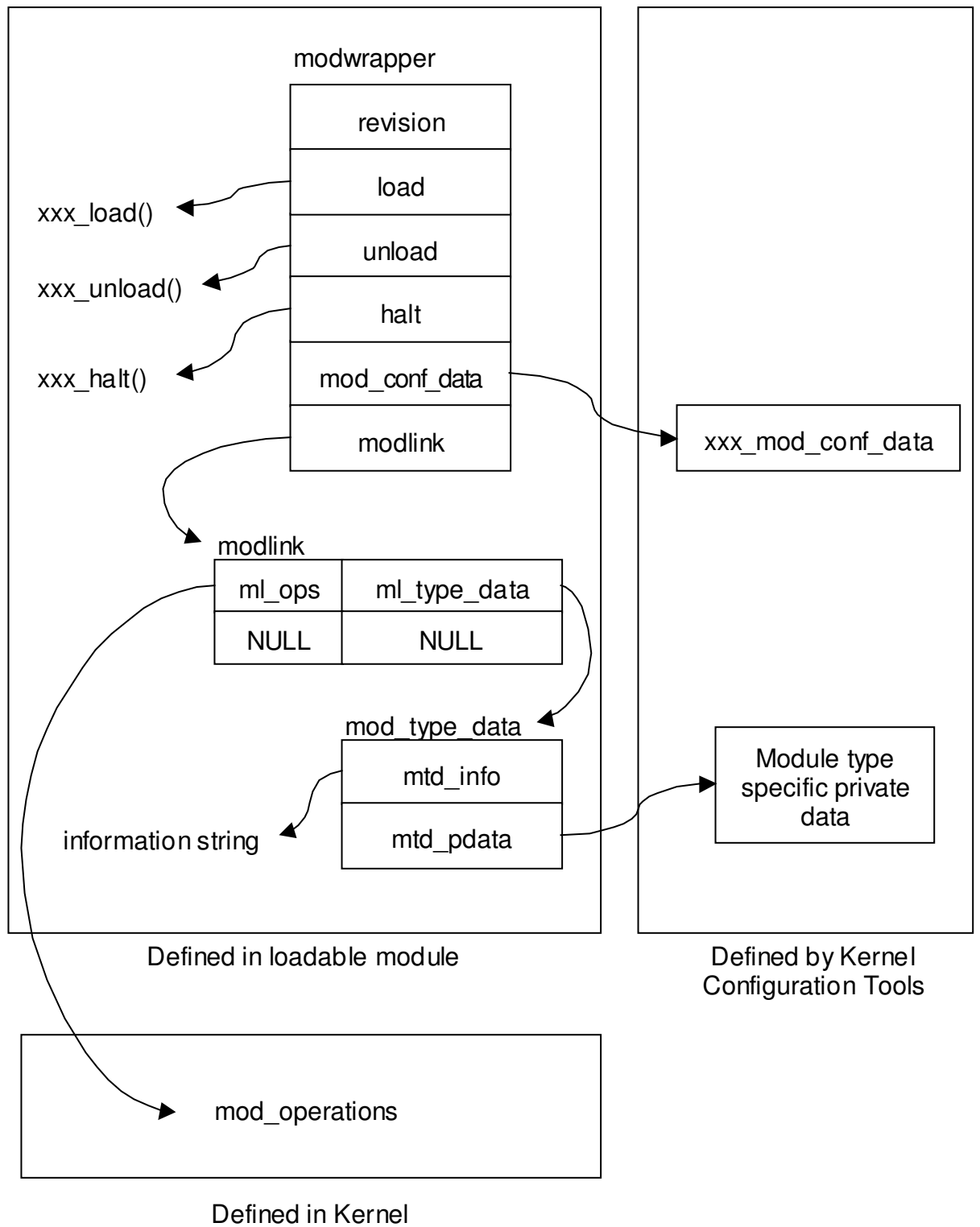
Modules can also be demand-loaded or linked statically into the kernel. "Demand-loading" means that the module will be loaded, usually by an administrator, prior to its use. It can be unloaded just as an auto-loaded module can be. When modules are linked statically, the kernel must always be rebuilt and the machine rebooted to change any file system code.

## Creating a File System Loadable Module

To create a file system loadable module several things need to be done. Routines must be added to the file system to support loading and unloading. Changes to the kernel configuration module metadata need to be made to enable the loadable module feature, and finally, the module needs to be placed in the proper location to be included in the running kernel.

### ***File System Code Changes:***

Adding the DLKM feature requires the addition of load and unload routines and a wrapper structure to the file system code. Figure 3 shows the relationship of the wrapper structure and the data structures required by the DLKM infrastructure. As can be seen, certain structures are defined in the loadable module while others are defined by the kernel and the kernel configuration tools.



**Figure 3: DLKM wrapper and related kernel structures**

The wrapper structure serves as an entry point used by the DLKM infrastructure to find the load and unload routines and other data within the file system code. Example 1 illustrates a sample DLKM wrapper and its required structures.

```

/* File system DLKM wrapper */
#include <sys/moddefs.h> /* Required header file */
#include <sys/mod_conf.h> /* Required header file */
#include <sys/mod_vfs.h> /* Required header file */

extern struct mod_conf_data cdfs_conf_data; /* Defined by kernel configuration tools */

/* Prototypes */
static int cdfs_load();
static int cdfs_unload();

/*
 * Required structures for DLKM infrastructure.
 */
static struct mod_type_data cdfs_mod_type_data = {
    "CDFS – Loadable CD file system", /* Descriptive character string */
    NULL /* Module type specific private data (kernel config tools controlled) */
};
static struct modlink cdfs_modlink[] = {
    {&mod_fs_ops, /* Type specific module operations */
    (void *)&cdfs_mod_type_data}, /* Kernel configuration data */
    {NULL, NULL}
};

/* Required DLKM wrapper structure */
struct modwrapper cdfs_wrapper = {
    MODREV, /* DLKM infrastructure revision number */
    cdfs_load, /* Module load routine */
    cdfs_unload, /* Module unload routine */
    NULL, /* Not supported. Must always be NULL */
    &cdfs_conf_data, /* Configuration specific data (kernel config tools controlled) */
    cdfs_modlink /* Type specific module operations */
};

```

### Example 1: Sample DLKM wrapper structures

The wrapper includes the required header files and prototypes for DLKM. Then follows the “mod\_type\_data” and “modlink” data structures containing information used by the DLKM infrastructure. The information in the “mod\_type\_data” structure should be changed to reflect your file system. The “mod\_conf\_data” structure is defined by the kernel configuration tools and contains information used by the DLKM infrastructure. The “modlink” data structure contains a pointer to the module type-specific operations that are used, by the infrastructure, to load, unload and perform other operations on this module type. The “modwrapper” data structure provides pointers to your file system-specific load and unload routines and links to the other required structures used by DLKM.

All DLKM modules must provide load and unload routines. These routines are simple and provide an entry point for the DLKM infrastructure to initiate the file system initialization routines. Example 2 illustrates a sample load and unload routine for an existing DLKM file system module. The wrapper also includes an entry for a halt routine. This routine is not supported and must always be null.

```

/* DLKM load and unload routines */
#include <sys/moddefs.h> /* Required header file */
#include <sys/mod_conf.h> /* Required header file */
#include <sys/mod_vfs.h> /* Required header file */

static int
cdfs_load(void *type_specific_data)
{
    /* Get module type specific private data – name and id*/
    fsmod_spec_t *cdfs_type_specific_data = (fsmod_spec_t *)type_specific_data;

    /* Install the cdfs specific vfs operations in the file system switch table */
    if(install_vfs(cdfs_type_specific_data->fsname, &cdfs_vfsops,
                  cdfs_type_specific_data->fs_id) == -1 ) {
        /* Installation in vfs switch table failed, fail load, return non-zero number */
        return(<errno from sys/errno.h>);
    }
    /* Call the file system initialization routines */
    cdfs_init();

    /* Always return zero on success */
    return(0);
}

static int
cdfs_unload(void *type_specific_data)
{
    /* Get module type specific private data – name and id*/
    fsmod_spec_t *cdfs_type_specific_data = (fsmod_spec_t *)type_specific_data;

    /* Free any allocated resources and unregister tunables here*/

    /*
     * Remove the cdfs specific operations and restore the auto load routine to the
     * file system switch table.
     */
    if(uninstall_vfs(cdfs_type_specific_data->fsname) == -1) {
        /* Uninstall failed, fail unload, return non-zero number */
        return(<errno from sys/errno.h>);
    }

    /* Always return zero on success */
    return(0);
}

```

### Example 2: Sample file system DLKM load and unload routines

In both routines the DLKM infrastructure passes file system-specific data consisting of the file system name and the file system identification number. Details on this data are provided below in the discussion of the module metadata file. In the load routine the call to “install\_vfs” populates the file system switch table with the entries required by the file system, in this case cdfs. If this routine fails, the load should be failed. Once the file system operations have been installed, the module can be initialized by calling the file system initialization routines.

When the unload routine is called, any allocated resources (i.e., memory, locks, etc.) should be de-allocated. The routine "uninstall\_vfs" is then called to remove the file system operations and restore the auto load routine in the file system switch table.

Remember, you are always required to include your file system install routine. This routine is used to install the file system if it is statically linked into the kernel. This should already exist in your file system.

#### File System Module Metadata

With HP-UX release 11i v2, new kernel configuration tools and procedures were introduced. One of these changes was the inclusion of a metadata file for each kernel component, which describes that component's characteristics. Example 3 shows a sample metadata file for a loadable file system.

```
/* CDFS module metadata file */
module cdfs {
    version 1.0.0          /* Module version number */
    desc "CD File System" /* Module description */
    type fileys           /* Module type */
    states auto loaded static /* Load states module supports */
    loadtimes run         /* When can module be loaded */
    unloadable           /* Module is unloadable */
    inifunc driver_install cdfs_install static /* Static initialization routine */
}

```

### Example 3: Sample file system module metadata file

As you can see, in this example the metadata file is written in a syntax similar to 'C'. The "version" field indicates the version of the loadable module. HP-UX (11i v2) supports full module versioning. New modules typically start with a version number of '1.0.0'. The "desc" field should include a descriptive title for the file system. The "type" field specifies the module type as a file system type. The "states" field indicates what states the module is capable of supporting. In the example, the module is capable of being auto-loaded, demand-loaded, and can be linked statically into the kernel. The first entry is the default capability. The "loadtimes" field specifies when in the boot process the module can be loaded. A value of 'run' indicates the module can be loaded at user space boot (generally when the 'rc' scripts are run) or anytime after that. If your file system can be unloaded, the "unloadable" keyword should appear. The "inifunc" field specifies when and how the file system should be initialized when it is statically linked into the kernel.

There are other fields and keywords that can be used in the metadata file. File system tunables would appear here and some file systems may have capabilities, not shown in the example, that would require different keywords. Consult the developer documentation for further details.

## Configuring a File System Loadable Module

Once the loadable file system has been built, it must be configured on a running system.

Bootable kernels and their associated data and modules reside in '/stand' on a running system. They are stored in directories corresponding to their configuration's names. This allows different kernel configurations to be stored in '/stand' under different names. The kernels loadable modules are stored within these configuration directories.

All new loadable modules must be delivered to '/usr/conf/mod' on the running system. The name used for your module must correspond to the prefix used in your wrapper data structure (e.g., 'cdfs' was the prefix used in the module wrapper data structures and the load and unload routines in our example. Our module name would then be 'cdfs' and delivered to this directory). The kernel configuration tools look in this location when configuring loadable modules. Once the new module is placed here the following sequence of steps will get the module configured in the running kernel (or another configuration if desired). Cdfs is used here as an example:

1. Unmount the file system.
2. Run: `kcmodule -c <configuration> -s cdfs=unused`
3. Run: `kcmodule -c <configuration> -s cdfs=auto`

The new module is now available in your configuration. If the configuration specified is the current booted configuration (or if the configuration specification is left out) the module is available for immediate use. Step 2 removes the module 'cdfs' from the configuration. This effectively means there is no cdfs in the configuration. Step 3 copies the new cdfs module from '/usr/conf/mod' and places it in the specified configuration directory. It also populates the file system switch table with a pointer to the module auto-load routine. If we wanted to load the module immediately we could use the 'loaded' keyword rather than 'auto'. We can also specify that the module be linked statically into the kernel by using the 'static' keyword. This requires a rebuild and reboot of the kernel.

## Planning File System Layout on Machines with Loadable File System Modules

Although loadable file system modules provide a great degree of flexibility, there are some points that need to be considered when configuring the file system layout on systems:

- o File system modules cannot be replaced without rebooting unless they can be unmounted. This means that if your '/root' or '/stand' file system type is the same type as the module you are replacing, you cannot do this without rebooting the machine.
- o Module dependencies can have an effect on whether a reboot is needed. For example, if another loadable module that cannot be unloaded has a dependency on your module, yours cannot be replaced without rebooting the machine.

When planning the file system layout on a machine, consideration should be given to having the '/root' and '/stand' file system a different type than the file system used to store the working set of data. For example, this is easy to accomplish on systems that have a specialized file system for a database application. The '/root' and '/stand' would be different than the data file system. The data file system module could be replaced without rebooting the machine. It would be unmounted, its module marked 'unused' and the new module copied to the system and configured into the running kernel configuration. The file system then only needs to be re-mounted.

This is also useful when debugging a new file system. One could configure the system so that the file system under test is not '/root' or '/stand'. This would significantly reduce debug time as you can replace the module with a new module without rebooting the machine. All standard debugging tools (e.g., kwdb, etc.) work with loadable file system modules and can be used for troubleshooting your code.

Itanium®-based systems support early boot loading of dynamically loadable kernel modules. This allows file system modules to be loaded during system initialization. File system types that are used for '/root' and '/stand' can be made loadable. However, replacement of these modules still requires a system reboot (but not a kernel rebuild).

## Summary

We have shown that with minor code changes and the new kernel configuration tools it is much easier to configure loadable file system modules into the kernel. With some planning of the file system layout, it is possible to avoid a time-consuming reboot when updating file system code.

## For more information

<http://www.hp.com/products1/unix/operating/infolibrary/whitepapers>

Kernel Configuration chapter of the Driver Development Guide, HP-UX 11i v2

## HP-UX 11i release names and release identifiers

With HP-UX 11i, HP delivers a highly available, secure, and manageable operating system that meets the demands of end-to-end Internet-critical computing. HP-UX 11i supports enterprise, mission-critical, and technical computing environments. HP-UX 11i is available on both PA-RISC systems and Itanium-based systems.

Each HP-UX 11i release has an associated release name and release identifier. The `uname (1)` command with the `-r` option returns the release identifier. The following table shows the releases available for HP-UX 11i.

HP-UX 11i releases

Release name	Release identifier	Supported processor architecture
B.11.11	HP-UX 11i v1	PA-RISC
B.11.20	HP-UX 11i v1.5	Intel Itanium
B.11.22	HP-UX 11i v1.6	Intel Itanium
B.11.23	HP-UX 11i v2	Intel Itanium

© Copyright 2003 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The warranties for HP 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. HP shall not be liable for technical or editorial errors or omissions contained herein.

Intel, Itanium, and Itanium Processor Family are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries and are used under license. UNIX is a registered trademark of The Open Group.

