AIX Hot-Patch Developer's Guide

Version 1.00
2/24/2009

This document provides an overview of the Concurrent Update (i.e., hot-patch) capability, as well as instructions for how to produce and manage hot-patches. Additionally, hot-patch procedures and limitations are provided.

In AIX 6.1 Technology Level 2 (TL2), the following support has been added for AIX Concurrent Update:
- ability to install a hot-patch over an existing hot-patch (i.e., removal of a prior hot-patch is not required)
- customer creation of hot-patches.

Trademarks

The following are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
IBM

Other company, product, or service names may be trademarks or service marks of others.

Terminology

concurrent update or **hot-patch** - a kernel or kernel extension fix that is applied to a running system and takes effect immediately. The fix can be applied or removed without rebooting. The fix does not persist across a system reboot.

applied hot-patch - an in-memory fix that does not persist across a system reboot.

committed hot-patch - an applied hot-patch that has had its update module written to the file-system. The fix is in-memory, and therefore active, but it will also persist across a system reboot.

ecfile - interim fix control file.

follow-on - a secondary patch for a module that has already been patched.

module - base kernel (`/unix`) or kernel extension or device driver.

reference module - original shipped module for the target system.

reference object - object file produced when building the reference module.

traditional or **conventional interim fix** - the interim fix approach used when a hot-patch is not available.

Table of Contents

Overview

A hot-patch is a special interim fix package that allows the base kernel and kernel modules to have program fixes applied, or subsequently removed, without the requirement of a reboot or other disruption to service. A hot-patch is created with the `epkg` command, and managed with the `emgr` command.

Not all fixes can be made available as a patch, since the patching mechanism has certain limitations. The requirements for hot-patching are given in the section **Hot-Patch Development Procedure**, below.

A hot-patch differs from the traditional interim fix package in that it contains one or more patch object files. The patch objects are used to perform in-memory patching of the base kernel and kernel modules. For each module to be patched there is one patch object. Additionally, the hot-patch package also contains one or more update modules.

A hot-patch applied for in-memory patching does not persist across a system reboot. However, the update module can be committed to the file-system to make the change persistent across reboots. Alternatively, the update module can be installed in the traditional manner, skipping hot-patching altogether.

During installation of a hot-patch, the hot-patch object file is read into system memory and linkage is performed. Linkage causes patch references to use the active data instances of the in-memory image of the specified kernel module. Function calls are redirected to the patched instance of the function. This is accomplished by overwriting the first instruction of the original function with a branch to the hot-patch version of the function. The memory into which hot-patches are loaded is pinned, and therefore is suitable for interrupt code.

Installation of a hot-patch for a module already having a patch (i.e., a follow-on) causes the prior patch to be effectively removed, although it is not removed from kernel memory. The follow-on should contain any prior fixes; otherwise, those fixes are lost when the follow-on is installed. Removal of the follow-on will reinstate the prior hot-patch, provided that the follow-on is not a committed hot-patch. Removal of a committed hot-patch reverts the module to its base state (i.e., no hot-patches). In the case of removal of a committed kernel extension hot-patch, further patching is disallowed for that module until a reboot is performed. For this reason, committing such hot-patches should be deferred until a reboot is imminent. Removal of a committed hot-patch for `/unix` does not face this limitation.

A kernel extension need not be loaded at the time a hot-patch is applied. Processing for an applied hot-patch is automatically completed at the time the corresponding kernel extension is loaded. (Note: developmental errors are easiest to detect when the kernel extension is already loaded; therefore, it is recommended to load a kernel extension prior to applying a hot-patch.)

Patch Object Creation

A hot-patch object file is created by compiling a program fix for the relevant source file. The program fix must meet certain requirements to be a valid hot-patch. Foremost of these requirements is that the change be limited to program text, or "code". That is, no changes can be made to either global or static data. Changes involving local variables (i.e., stack data) are permitted, provided static data is not involved.

Furthermore, each fix must be limited to a single function. While a single hot-patch may fix multiple problems for a given module, each fix must be valid when applied independently to the system. See **Appendix A - Hot-Patch Fix Requirements** for a complete list of the requirements. A traditional (i.e., non-hot-patch) interim fix must be used in cases where a fix does not meet hot-patch requirements.

Static Patch Checking

The `patchchk` command exists to assist in verifying that compile-time limitations for hot-patch have been honored. Most violations can be identified by the command, though not all. The `patchchk` command works by examining the symbol table, relocation data, and program data within the patch object, the reference object and the reference module. The command is located in `/usr/bin`.

Each object used for hot-patch creation must be verified individually (i.e., prior to linking with `-r`, if required). Running the `patchchk` command on an object which has been linked with `-r` will result in erroneous errors.

Errors from the `patchchk` command are documented in **Appendix B - Explanation of Patchchk Errors**. The `patchchk` command's error messages typically refer to symbol table entries. The dump command can be used to examine symbol table entries:

```
$ dump -X64 -tv file
```

If the patch developer is unable to correct errors reported by the `patchchk` command, a hot-patch cannot be created. The problem must be serviced with a traditional interim fix.

Note: The Visual Age C Compiler, under certain conditions, creates the following special symbols: `_$STATIC`, `_$STATIC_RO`, and `_$STATIC_BSS`. Static data is collected to one of the symbols, and compile time offsets are then created to reference the data relative to a base pointer for the given area. The hot-patch object must have identical static data layout, so that it matches the actual data in use by the module. A reference to one of these symbols within a `patchchk` error indicates that static data of some kind was modified, whether due to additional, rearranged, or deleted static data.

Some examples of non-obvious constants are as follows:

- use of defines new to the file
- date-stamps, perhaps by a compile line define, or via checkout from a source code version control system
- macros
- use of `__FILE__`, `__LINE__`, etc..

Package Creation

A hot-patch is created with the `epkg` command. The interactive mode of `epkg` cannot be used for hot-patch package creation; therefore, an `epkg` control file (ecfile) must be used. Care must be taken to ensure that each stanza within the control file is set correctly. Several errors will only be evident when installing the patch.

If multiple hot-patch object files exist for the same module, they must be linked into a single relocatable object file prior to packaging. This is accomplished by linking the objects with the `-r` flag.

An example of the ecfile for a single file change to `/unix` is shown below (Note that comments on the right-hand side are for documentations purposes only. The `epkg` command only supports comments within column 1.):

```
--------------------------------------------------------------------------------

# Interim Fix control file example
ABSTRACT=This is a test of epkg with concurrent updates.
REBOOT=yes                          # emgr will bosboot if a commit is done
DESCRIPTION=/tmp/description        # Loc of text file containing descript.
# CU_VR=no                          # Doesn't require V=R addressability
PRE_INSTALL=.
POST_INSTALL=.
PRE_REMOVE=.
POST_REMOVE=.
PREREQ=.
EFIX_FILES=1                        # Just one patch object in this example

EFIX_FILE:
    EFIX_FILE_NUM=1                 # One file in this patch
    ACL=root:system:755             # Perms: root:system:755
    INSTALLER=1                     # 1 is for installp managed files
    TYPE=3                          # Type 3 indicates it is a Conc. Update
    SHIP_FILE=/home/test/kext       # Loc to get the module binary from
    TARGET_FILE=/usr/lib/boot/kext  # Install location
    CU_OBJECT=/home/test/my_patch.o # Loc to get patch obj from
    CU_LDR_NAME=/usr/lib/boot/kext  # Name which module is known by
    CU_MODULE_MEMBER_NAME=          # Used if kext with ar member
    SANDBOX_BINARIES=/sandboxes/gold/kext # Loc to get module timestamps
    AR_MEM=.

--------------------------------------------------------------------------------
```

The stanzas pertinent to Concurrent Update are described below:

`CU_VR` – When set to `yes`, the patch receives V=R addressability. Most patches do not require V=R addressability. (Commented out above, and may be omitted when not required.)

`EFIX_FILES` – Set to `1`, signifying a single module is patched.

`EFIX_FILE` – Marks the stanzas which describe the patch.

`EFIX_FILE_NUM` – Number of modules patched

`INSTALLER` – Set to `1` if the file is `installp` tracked; otherwise it is set to `5` (not `installp` tracked).

`TYPE` –Must be `3` for hot-patch

`CU_OBJECT` – The patch object's full pathname (absolute path)

`CU_LDR_NAME` – The absolute pathname under which the module is loaded from, beginning with '`/`'. (The base kernel is an exception in that it is referenced as `/unix`.)

Note: The `/unix` symbolic link must point at the kernel which the system was booted with. If this is not the case, attempting to apply a Concurrent Update will fail with an error.

`CU_MODULE_MEMBER_NAME` – The archive member name, in the case where a kernel extension is an archive. Otherwise unset.

SANDBOX_BINARIES – A colon separated list of reference binaries nominally including a single reference module. Typically, it is necessary only to specify a single reference. However, if a patch object is valid for multiple versions of reference modules they would be specified using the colon separated syntax.

AR_MEM – Kernel extension archive member name. Must be set to . (period) if no member exists.

Once the control file has been created, the epkg command is used to create the hot-patch interim fix package. The ecfile and a label must be specified:

```
$ epkg -e ecfile label
```

where *ecfile* is the filename of the interim fix control file you just created, and *label* is a unique label name chosen by you for the package. The name of the output file is printed as a result of executing the above command. It will reside in your home directory under epkgwork/label_name/*.Z.

Once the interim fix package is created, it must be moved the system that is to be patched.

Hot-Patch Installation

A hot-patch should be installed on a system having the appropriate level of module to be patched. The emgr command is used to install the hot-patch in-memory:

```
$ emgr -i epkg_path
```

Hot-Patch De-installation

To de-install a patch, use the emgr -r flag, with the label:

```
$ emgr -r -L label
```

Committing a Patch

It is recommended to only commit a hot-patch update module just prior to a reboot. Removal of an in-memory committed patch causes all patches for the kernel module to be removed. Furthermore, additional hot-patching for the kernel module, other than /unix, is disallowed.

Committing a hot-patch is only permitted when an in-memory hot-patch is either already in place, or if the commit is specified in conjunction with an in memory installation.

If a hot-patch (in-memory) is in place, the update module for it can be committed to the file-system as follows:

```
$ emgr -C -L label
```

Alternatively, the commit can be issued in conjunction with the in-memory installation of the hot-patch, but is discouraged due to the previously mentioned limitations. The following emgr command installs a hot-patch in memory and commits its update module to the file-system:

```
$ emgr -C -i epkg_path
```

Listing Patches

The hot-patches that have been installed on the system are listed with the `emgr` command's `-l` flag. If a follow-on patch has been installed, the hot-patch that it automatically deactivated has the follow-on's label listed in the `UPDATED_BY` column.

Patch Limitations

Constants that are 32-bits or larger cannot be put into a hot-patch, since the compiler places them within the static data area for the file. Certain constants, smaller than 32-bits, may also be placed within the static data area. Use the `patchchk` command to detect such instances.

Follow-on Creation

A hot-patch created for a module already having a hot-patch is termed a *follow-on hot-patch*, or just *follow-on*. When a follow-on hot-patch is installed, the prior hot-patch is automatically unpatched. Therefore, follow-on hot-patches must include any prior hot-patches for a given module. A special case would be when an existing hot-patch is not needed, and it is to be intentionally omitted by the follow-on.

Hot-patches requiring multiple object files, such as is likely in the case of a follow-on, must be linked as a single relocatable object (`ld -r`).

Common Problems

Timestamp mismatch - This error will occur if the wrong module is specified to `SANDBOX_BINARIES`. One common reason would be mistakenly specifying the update module in place of the reference module. Care must be taken to correctly identify the reference module. The XCOFF timestamp is used for the timestamp comparison, and can be displayed with the `dump` command:

```
$ dump -X64 -ov file
```

Multiple versions in memory - Hot-patch requires that only a single version of the module be loaded. Attempting to hot-patch in the presence of multiple versions of a loaded module results in a multiple versions error. A common way this could occur would be on a system where kernel module development is being performed. In development care should be taken to unload after each load of the kernel module. Additionally, the `slibclean` command must be run after each unload. The `genkex` command can be used to determine if multiple occurrences of a module have been loaded (although it won't indicate if they are of the same version or not).

Module not patched after installing the hot-patch - A hot-patch will not have effect if the path specified by the ecfile's `CU_LDR_NAME` stanza does not match the path by which the module was loaded. The `CU_LDR_NAME` stanza must match the load path. Such instances consume space within the patch memory of the operating system.

System Patch Limits

The design of hot-patch is such that the patch space cannot be recovered when a hot-patch is removed. Although the patch space is many mega-bytes in size, repeated development and installation of hot-patches can exhaust the available patch memory. The system must be rebooted in order to recover the patch memory.

Hot-Patch Development Procedure

Methodical development and deployment practices are required for the successful installation of a hot-patch. For this reason the following procedure is prescribed:

Preliminary Requirements

1) Uniqueness of module filenames.
   All sources files composing hot-patchable module must have filenames that are either unique, or be unique within the last three directory path components. The file pathnames are recorded within the XCOFF symbol table and can be examine using the `-tv` options of the `dump` command. An example of source files that do not meet this requirement follows:

   ```
   src/x/b/c/this_file.c
   src/y/b/c/this_file.c
   ```

   Since the last three components of the pathname are not unique, the module comprising them is not hot-patchable.

   Object files built within the same directory as the sources will not contain any directory components. Provided that no other source file for the module has the same name, this meets the requirements for hot-patch.

2) Compilation requirements.
   Reference modules must be built with Visual Age C (VAC) compiler version 9.0.0.1 or greater.

3) Retention of sources, objects, and build environment.
   Hot-patch creation requires that all source, including makefiles, program source, header files, and all resulting object files, as well as the shipped module be retained for a module to be hot-patchable.

   Source code control system identifiers within fix files must be identical to the original file. Some control systems fill in a data initialization composed of a string initializer, for instance, which may contain a date-stamp, or some other information which varies. All source files composing the fix must not contain such alterations.

4) XCOFF symbol table (not to be confused with the `.loader` section symbol table).
   A module to be hot-patchable must retain its XCOFF symbol table. (i.e., The XCOFF symbol table must not be stripped.)

5) Target system must be at a supported level.
   Target here refers to the system where hot-patch installation will occur. Nominally, this requires a system at 6.1 TL2 or later.

6) On the target system, the module must not have conventional interim fixes applied.

Selection and Fix Requirements

See **Appendix A - Hot Patch Requirements**

1) Corruption issues, whether within memory or within a file-system, are not suitable for hot patch. A corruption that has already occurred will not be remedied by a fix.

Build Requirements

1) The Visual Age C compiler version 9.0.0.1 or greater must be used and the `-qxflag=patch` compiler option specified when building hot patch objects. Some optimization flags can rearrange data layout. It is recommended to use `–O2` or less.

Packaging Requirements

1) Hot-patches must be packaged on a system installed with AIX 6.1 TL2 or later.
2) An `epkg` control file (ecfile) must be created and used to package a hot-patch (i.e., interactive mode is not supported for hot-patches). The appropriate hot-patch stanzas must be supplied within the `epkg` control file. See the **Package Creation** section above.

Verification Requirements

1) Each object included for a hot-patch must be successfully verified using the hot-patch checking command `patchchk`. If any errors are detected, the fix must be altered so that the errors are resolved. If the `patchchk` command issues an error that cannot be resolved, the fix is not suitable for hot-patch.

2) Install the hot-patch, as an in-memory only fix, on a test system at the proper level and verify that the remedy is operational. This verification step may be performed using a unit or functional verification test, or by whatever other means deemed necessary.

3) Uninstall the hot-patch, reinstall it, and verify with whatever method was used in verification Step #2. This verifies that the design of the fix considered reinstallation, and repeatability.

4) Commit the hot-patch to the file-system, reboot, and verify with whatever method was used in verification Step #2. This verifies that the update module contained within the hot-patch package is correct.

List of Hot-Patchable AIX Kernel Extensions

In addition to `/unix`, the hot-patching is supported for the following kernel extensions:

AIO, LVM, NFS, Planar PAL, Virtual Bus, VSCSI, and the following SCSI variants:
- Emulex FC adapter driver
- FC/iSCSI/SAS disk driver
- Parallel SCSI disk driver
- FC DS4K driver
- Emulex FC SCSI protocol driver
- Qlogic iSCSI TOE protocol driver
- iSCSI SW solution driver
- Qlogic iSCSI TOE adapter driver
- SIS SAS adapter driver
- LSI SAS adapter driver

Notes on kdb

Command `kdb` and kernel KDB both display patch text symbols prefixed by a pound sign, '#'. In the following example, `.kext62a` has been patched:

```
(0)> dis .kext62a 2
#.kext62a+000000        li    r3,A
#.kext62a+000004        blr                          r3=000000000000000A
```

There is no provision for accessing the original (non-patched) symbol by name.

Appendix A - Hot-Patch Requirements

One of the most important restrictions for hot-patch is that no changes be made to static or global data. There are many ways that static data, in particular, can be modified that are not self-evident. For instance, strings and literal data constants (depending on size) are static data. Additionally, switch statements are implemented as a table of addresses within static data. Many of the requirements listed below identify various ways that program data could be inadvertently modified.

The `patchchk` command should be used to verify that a hot-patch has not violated the requirements. However, not all violations of the requirements can be identified by it, and the hot-patch developer must ensure that the following requirements have been satisfied.

1. The fix must be built using the Visual Age C (VAC) Compiler version 9.0.0.1 or greater, specifying the `-qxflag=patch` compiler option.

   VAC version 9.0.0.1 and greater supports a new flag, `-qxflag=patch`, which must be specified when building a hot-patch object. This flag is used in addition to the flags specified for the creation of the reference object.

   All other compiler flags must be identical to the reference build. It is important that no other alterations be made to the flags, or the resulting hot-patch could be invalid. It is recommended that modules be built using an optimization level of `–O2` or less for hot-patch.

2. The fix must consist only of changes to the base kernel and kernel extensions that are not stripped. (The AIX kernel extensions ready for hot-patch are listed in the detailed kernel extension list below).

   For kernel extension patches, the kernel extension must be shipped with its XCOFF symbol table present (i.e., un-stripped). Stripping of the symbol table is typically accomplished either by use of the `strip` command, or by specifying the `-s` linker flag at link time.

   The XCOFF symbol table is different from the `.loader` section symbol table. To verify that the XCOFF symbol table is present, execute the following:

   ```
   $ dump -ov kext
   ```

   where *`kext`* is the kernel extension of interest.
   If the "`Symbol Ptr`" field is zero (or "`# Symbols`" is zero), the file has no symbol table.

3. The source fix must be to C code.

4. The fix must be only to program text and/or local (stack) data. Non-local data definitions cannot be modified in any way (i.e., stack variables may be altered, but extern and statics may not). Initialization values must not be change from the original's values, and data layout must not be affected.

   This includes the following:
   - no additions, deletions, rearrangement, or other modifications of data definitions, including strings
   - no modifications to a structure’s layout including those resulting from type changes
   - no new long constants, for which the compiler creates static entries.

   Examples of changes that are disallowed:
   - addition of a new string

- altering a string (Ex. change `"thsi"` to `"this"`, or modification of the SCCS id string)
- changing the dimensions of a static or extern array
- changing a data structure
- changing the data type such that its size changes
- changing an existing variables storage location by altering the compiler options: `-qdatalocal` or `-qdataimported`.

Examples of changes that are allowed:

adding, removing, or changing automatic variables, including alterations to their types, an array's dimensions, or initializations (excluding character strings, and large constants – i.e., those that reside within a compiler generated static data area).

5. No new function descriptors. Adding a function descriptor modifies data. A typical way to cause the creation of a function descriptor is by taking the address of a function, which previously didn't have its address taken.

   New functions can only be created if they will not require a function descriptor. (Functions typically do not require a descriptor if they are not exported and do not have their address taken.)

6. Functions cannot be moved from one file to another.

7. Function return type and parameter types cannot be altered (i.e., cannot be added, removed, rearranged, or modified in type so as to modify their sizes). Technically, an existing parameter's type could be changed to another type provided that the two are equivalent (i.e., have the same size, and are either both signed or both unsigned).

8. Changes to more than one function must only be done if those changes maintain integrity when done independently (i.e., applied serially to the system in any order). If a fix requires two or more simultaneous changes between functions, it is not suitable for hot-patch. This same restriction applies to recursive functions. However, two separate fixes (i.e. two different defects) to two different functions can be made into a hot-patch.

   The rationale for the above limitation follows. If the function to be patched already has a thread of execution present within it at the time patching occurs, and then subsequently calls another function, which also is patched, there will be a mismatch (potentially) between the two functions. Consider a function pair where one function sorts data into a forward order, and the other processes it, expecting such an ordering. While a new function pair, which uses a reverse sort order, would be self consistent, it would produce incorrect results if a thread of execution was already in the producing function at the time the new functions were patched in. When the consumer function is called, the new version will be invoked, which will mismatch with the expected data ordering. A patch to a recursive function would require close scrutiny along these lines, as would any function which may branch back to the first instruction of the function.

9. No modifications to exports are allowed. Additions to exports, even if the function or variable is not new, are not allowed. Deletions are not allowed.

10. Switch statements cannot have cases added, deleted, or rearranged.
    Switch statements, greater than a nominal (compiler dependent) size, cause creation of a table of addresses within the static data area. To work around this limitation, `if`/`else` logic can be inserted prior to the switch to contain the necessary changes.

11. New switch statements cannot be created and switch cases cannot be reordered or renumbered. Use `if`/`else` logic to work around this limitation. (Switch statements are implemented as a table of addresses within static data.)

12. References to imported symbols are restricted to those which the reference module already references. No new imported references are allowed.

13. Since patches can only be applied once the system is initialized, only code which is either active or could become active (in the case of a kernel extension load) after system initialization should be patched.

14. Code which requires V=R (virtual addresses equal to real addresses), must specify `CU_VR=yes` in the ecfile. The patch space for V=R patches is 128KB, and most hot-patches should use the normal patch area. This is accomplished by either specifying `CU_VR=no`, or by omitting `CU_VR` from the ecfile.

15. Kernel threads may be active within any version of the code at any time, whether the hot-patched version or the original code. There are various means by which this may occur, including registration of a function, or via `setjmp`, or other such methods. This must be considered when choosing code fixes for hot-patch.

Appendix B - Explanation of Patchchk Errors

This document provides a list of errors which the `patchchk` command can produce. A description is provided for each possible error, along with example `patchchk` output. Corrective action, where possible, is also specified. It is beneficial to have an understanding of XCOFF, since most errors contain XCOFF-specific information. If you are not familiar with XCOFF, see the XCOFF reference pages.

Most errors identify the symbol for which the error is issued, possibly with other information such as the symbol table index (for the patch or reference object/module), the file or other associated symbol table information. In the following examples, such literals are shown in italics, since they are specific to the actual values pertinent to the error message.

Some understanding of compiler specifics may be necessary to have the fullest understanding of some errors. One such example is the compiler's generation of the symbol `_$STATIC`. The compiler creates the special symbol `_$STATIC` when static data exists in the compilation unit (the file plus all its includes). If a source file to be patched had no static data, the symbol `_$STATIC` will not exist within its symbol table. If the patch contains an new string, the error "`Extra patch object symbol definition`" will be issued for `$STATIC`. Some errors are only possible from assembler code, and are so noted. Errors resulting from a non-conforming XCOFF file are also detected, and are so noted.

1) A symbol table entry in the patch refers to an unknown section type (non-conforming object).

```
error: Unexpected section type flag 0x4 for section 2.
```

To correct: Use a known section type, for text, data or bss.

2) A data definition has been added in the patch (whether initialized data or uninitialized).

```
error: Extra patch object symbol definition.
       Symbol: symname
       Definition in patch object does not exist in the reference object.
       (index 24 in patch object).
```

To correct: Do not add data definitions in a patch. (This includes strings.)

3) The filename of the patch differs from the reference object's filename. This is based on the `.file` entry within the XCOFF symbol table.

```
error: Missing symbol definition.
       Symbol: .file
       Definition in reference object is missing in patch
       (index 0 in reference object).
```

To correct: The filename for the patch must not be changed from the original (reference) source file name.

4) A symbol table entry for the patch specifies a storage class different than the reference object (assembly code required).

```
error: Storage class mismatch.
       Symbol: symname
       Object        SymNdx     Storage_Class
       patch             24        extern
       reference         24        hidext
```

To correct: The patch must use the same storage class for the symbol as it has in the reference object.

5) A change has been made which affects a symbol's storage mapping-class.
One such example would be the combination of using the `-qroconst` compiler flag with `const` variables in the patch.

```
error: Storage mapping class mismatch.
       Symbol: symname
       Object        SymNdx     SM_Class
       patch             24        RO
       reference         24        RW
```

To correct: The patch must use the same storage mapping class for the symbol as it has in the reference object.

6) A definition is changed from initialized data to unitialized data. The storage mapping type changes from section definition (`SD`) to common (`CM`).

```
error: Storage mapping type mismatch.
       Symbol: symname
       Object        SymNdx     SM_Type
       patch             24        CM
       reference         24        SD
```

To correct: The patch must not change data definitions.

7) A symbol differs in the number of auxiliary symbol table entries it requires (assembly code required).
An auxiliary entry contains extra information for a symbol table entry.

```
error: Number of auxiliary entries mismatch.
       Symbol: symname
       Object        SymNdx     Num_Aux
       patch             24        2
       reference         24        1
```

To correct: The patch must not alter auxiliary information.

8) A symbol in the patch has a different section number than in the reference object (assembly code required).

```
error: Section number mismatch.
       Symbol: symname
       Object        SymNdx     Sect_Num
       patch             24        2
       reference         24        1
```

To correct: The patch object must not move the symbol between sections.

9) A csect in the patch has a different length than what is in the reference object. (A csect is an XCOFF "Control Section", and is used to contain program text or data). One way this can occur is that a variable's type changed in size, say from `char` or `short` to `int`, or vice-versa. Another way is that static data was added or removed in the patch. In the later case, the symbol `_$STATIC` will be the symbol with the mismatch.

```
error: csect length mismatch.
       Symbol: symname
       Object        SymNdx     cslen
       patch             28       0x1
       reference         24       0x4
```

To correct: The size of data symbols must not change. Static data may not be added, removed, or modified in any way.

10) A data definition has its initial value modified.

```
error: Initialized value mismatch.
       Symbol: symname
       Object        SymNdx
       patch             24
       reference         24
              Start of Differing Bytes
                    Patch  Reference
       byte   3:  03  ''  02  '' <<<<
       byte   4:  00    '  00    '
       byte   5:  00    '  00    '
       byte   6:  00    '  00    '
```

To correct: Data symbol initialization must not change.

11) A function contains a branch absolute at its entry point (assembly code required).

```
error: Branch absolute at entry of function.
       Symbol: func_has_ba
       Object        SymNdx
       patch             16
```

To correct: Function entry points must not contain a branch absolute. Reference modules having a branch absolute at a function entry cannot be patched.

12) A new function was added to the patch and its address taken.

```
error: Address taken of new function.
       Symbol: symname
       Object        SymNdx
       patch             34
```

To correct: The patch may only take the address of a function whose address is taken in the reference source.

13) The patch's source filename, as specified by the patch's symbol table `.file` entry, has no corresponding entry within the reference module's symbol table.

```
error: Module does not contain the specified file entry within its symbol table.
       Module: /unix
       File:   foo.s
```

To correct: The patch must be built from a source file having the same name as was used for the reference object.

14) The patch object contains relocation entries for the TOC, but does not contain a TOC symbol in its symbol table (assembly code required).

```
error: Patch object has R_TOC relocation, but no TOC symbol.
```

To correct: Add a proper `.toc` pseudo-op to the assembler file.

15) Missing `.loader` section in module (non-conforming XCOFF).

```
error: Module has no loader section.
```

To correct: This is a sanity check, there is no recourse.