mxload

A Portable Package For
Reading Multics Backup Tapes

User’s Manual

mxload Release 1.0

1 December 1988

INTRODUCTION

The **mxload** package is a set of programs for interpreting data from backup tapes created on a Honeywell Multics system (using the Multics backup_dump command) on a wide variety of target systems. Throughout this manual, “the **mxload** package” refers to the entire set of programs, whereas “**mxload**” alone refers to the reloading program itself. The rest of this manual contains the following sections:

Data Formats and Conversions
: Describes how **mxload** treats the data it reads from tape, how it converts ASCII and binary files, and how it treats special Multics data formats (mailboxes, archives, Forum meetings).

Using the **mxload** Package
: Describes the programs making up the package and how they are used. Also describes naming conventions for reloaded data and the format of the reload map.

Using the Standalone Utilities
: Describes the standalone programs for manipulating Multics archives, mailboxes, and Forum meetings.

Control File Syntax
: Describes the control file used to specify options, pathnames, and name translations for batch-style use.

Installation
: Describes files and directories on the **mxload** distribution tape and how to install them.

Release Notes
: Miscellaneous notes on the current release of **mxload**.

Typographic Conventions

In the text of this manual, use of **bold** font identifies UNIX commands, filenames, and, in general, any string interpreted or produced literally by a target system. The *italic* font is used to indicate parameters which must be replaced before use. The listing font (in the text) indicates a Multics command, subroutine, pathname, and, in general, any string interpreted or produced literally by Multics. In examples (set off from the text), the bold listing font is used for user input (to the target system, or, sometimes, to a Multics system). When an example includes expected output, that is shown in listing font.

Wherever possible, this manual does not assume any particular target system. However, since many features (such as user ID translation) are specific to UNIX systems, the SunOS version of **mxload** is used in all examples. When running on other (non-UNIX) systems, the commands may have different syntax, file names may be different, etc.

UNIX is a registered trademark of AT&T Bell Laboratories. mxload is a trademark of Oxford Systems, Inc. Multics is a registered trademark of Honeywell Bull, Inc. MS-DOS and PC-DOS are registered trademarks of Microsoft, Inc. SunOS is a trademark of Sun Microsystems, Inc. Sun-2 and Sun-3 are registered trademarks of Sun Microsystems, Inc.

DATA FORMATS AND CONVERSION

mxload and its standalone utilities are primarily in the business of converting from Multics data formats to some form usable on the target system. This involves both simple conversion from 9-bit Multics bytes to 8-bit bytes, and complex unpacking and conversion for special Multics formats such as archives and mailboxes. This section discusses the details of such conversion, but the actual conversion specifications are described in *Control File Syntax*, below.

Data Formats

Because Multics data is composed of 9−bit bytes, and **mxload** is designed for systems using 8−bit bytes, Multics data must be converted to an appropriate corresponding form as it is reloaded. For byte-oriented data, such as ASCII text, each Multics byte is converted to a corresponding byte in the new file; otherwise, the conversion is bit-oriented. In addition, **mxload** recognizes several special Multics data formats (such as mailboxes and archive segments) and, by default, converts them automatically into an appropriate form when reloading. Statements in the **mxload** control file can override any of the default conversions.

mxload recognizes two basic formats for Multics data: 9−bit binary and 8−bit ASCII[1]. When reloading Multics files (“segments”, in Multics terminology), **mxload** examines the file’s contents to determine which format is appropriate. In general, if none of the 9−bit bytes in the original Multics data has the high-order bit set, the data is treated as 8−bit ASCII, and the low order 8 bits of each 9−bit Multics byte are reloaded into an 8−bit byte in the reloaded file. This is specified as **8bit** conversion.

If the data is not 8−bit ASCII ASCII, it is treated as 9−bit binary: the 8 high-order bits of the first Multics byte are reloaded into the first byte of reloaded data; the second byte of reloaded data has as its high order bit, the low-order bit of the first Multics byte, and as its 7 low-order bits, the 7 high-order bits of the second Multics byte, etc. This is a “big-endian” view of byte ordering, and converts each string of 8 Multics bytes into 9 8−bit bytes. This is specified as **9bit** conversion.

9−bit binary conversion is best illustrated by the following diagram:

Multics Data (Eight 9−bit bytes)							
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
$2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$

$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$	$2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$
Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Reloaded Data (Nine 8−bit bytes)								

In addition to being converted in this bit-oriented format, 9−bit binary data can also be converted to 8−bit ASCII format by stripping the high-order bit from every byte. This can be performed manually after a file has been reloaded (see the **mxascii** manual page in the appendix) or by an appropriate **convert** statement in the **mxload** control file. This is specified as **8bit** conversion.

[1] Strictly speaking, 8−bit ASCII data is not ASCII, since the ASCII character set includes only those with values from 0 to 127 (decimal). However, it is a convenient term, and throughout the **mxload** documentation, the term “ASCII” refers to 8−bit ASCII (byte-oriented) data, and “binary” refers to 9−bit binary (bit-oriented) data.

The “primitive” conversion types are **8bit**, **9bit**, **8bit+9bit**, and **discard**. The **8bit** conversion type converts byte by byte, discarding high-order bits as needed. The **9bit** conversion type converts in bit-stream form, as above. The **8bit+9bit** conversion type produces two files for each Multics input file: one in **8bit** form, the other in **9bit** form, distinguished by the suffixes **#8b** and **#9b**. Finally, the **discard** conversion simply discards the Multics data, useful when it is clear that there will be no use for keeping it (such as Multics object segments). The **mxascii** standalone utility may be used to force **8bit** conversion on a file reloaded as **9bit**; the opposite direction is not possible, of course.

Special Multics File Types

The following Multics file formats are supported either by **mxload**, by the standalone utilities, or both: archive segments, mailbox segments, and forum meetings. Multics multi-segment files and non-ASCII files may also be manipulated after reloading.

In normal operation, **mxload** automatically unpacks any archive files that are entirely 8-bit ASCII into corresponding directories containing all the archive’s components but reloads the other special formats, including archives with any components requiring **9bit** conversion, as **9bit** data. Archives are unpacked recursively: if an archive component is itself an archive, it, too, is turned into a directory and unpacked.

When requested for **mxload**, or when performed standalone by **mxmbx**, mailboxes are unpacked into a file or files containing the individual mail messages. Several options are available for this conversion; see **mxmbx** for details. When a mailbox is unpacked, special header lines are prefixed to each message to indicate the original sender, access class, and time of the message (i.e., the data from the mseg_return_args structure). For efficiency, **mxload** by default simply reloads mailboxes as files and requires that they be unpacked manually by **mxmbx**.

Multics Forum meetings are reloaded as UNIX directories by **mxload**, but must be unpacked manually afterward. This is necessary because a Forum meeting (or any multi-segment file) may be split across two tapes, and **mxload** has no way to determine that the complete meeting has been reloaded. Options similar to those for mailboxes are available for this conversion; see **mxforum** (in the appendix) for details.

Multics multi-segment files (MSFs) are also reloaded as UNIX directories by **mxload**, but no automatic processing is available. An MSF created by vfile_ in stream_output mode (such as by file_output or crossref) can be turned into a UNIX file by using the **cat**(1) command to concatenate its components.

A non-8bit file reloaded as 9−bit binary may be converted by stripping off the high-order bit of each byte using the **mxascii** utility.

USING THE MXLOAD PACKAGE

There are three basic operations that can be performed by the programs comprising the **mxload** package: reloading a tape, listing a tape's contents, and unpacking or converting data already read from a tape. These operations are performed by the following programs:

mxload	Reload specified files and directories from a backup_dump tape (or file image).
mxmap	List the contents of a backup_dump tape (or file image), optionally displaying all Multics segment and directory attributes, names, ACLs, etc.
mxarc	For a Multics archive file already reloaded with **mxload**, extract components, list the contents, or unpack into an archive directory.
mxmbx	For a Multics mailbox segment already reloaded with **mxload**, unpack into a file or files containing all the individual messages.
mxforum	Like **mxmbx**, but unpacks a Multics Forum meeting (reloaded as a directory and several files) into a file or files containing all the transactions from the meeting.
mxascii	For a file already reloaded with **mxload**, converts from 9-bit binary format to 8-bit ASCII form.

The primary function of the **mxload** package is reloading Multics data, and most of this section describes how **mxload** itself is used. The next most important program is **mxmap**, which is described following **mxload** (see *Reload Map Format*, below, and the **mxmap** manual page in the appendix). Finally, **mxarc** through **mxascii** are described as a group (see *Using the Standalone Utilities*, below).

Complete interface descriptions of each program are provided by the UNIX-style manual pages in the appendix at the end of this manual. The descriptions here are intended as examples and general guidance, not as a complete reference.

Styles of Use

mxload supports two different styles of use: interactive and batch. The "interactive" style, where **mxload** is controlled primarily from command line options, is intended for simple reloads of entire subtrees, where the standard conversion defaults are satisfactory and the user name conversion is not important.

In the "batch" style, **mxload** is controlled by statements in one or more **mxload** control files. This is intended for selective reloading, where an entire Multics hierarchy, containing data belonging to many users, is reloaded into a corresponding hierarchy on the target machine. For batch operation, the control files can be used to specify translations from Multics identities to appropriate owners on the target machine, and to select multiple subtrees (or individual files or directories) for reloading into specific new locations.

In both styles of **mxload** (and **mxmap**) use, only one tape is processed at a time. When a tape is finished, the **mxload** command terminates; a new tape must then be mounted, and the command issued again, if more data is to be reloaded. Because UNIX systems lack a standard tape handling interface, it is not possible for **mxload** to handle multiple tapes itself. Instead, this must be accomplished by invoking **mxload** repeatedly.

By default, **mxload** reads the entire tape looking for data to reload. The **−f** option[2] may be used to cause **mxload** to exit immediately after it has reloaded the requested data, rather than looking for additional copies later on the tape. This option should be used when reloading specific objects from a tape, rather than the tape's entire contents, as otherwise, **mxload** will read the tape through to the end looking for additional copies even after satisfying the requests for reloading specific objects.

WARNING: **mxload** is intended primarily for use on "complete" backup_dump tapes. Attempting to reload an "incremental" or "consolidated" tape will create multiple copies (with different names; see *Reload Name Translation*, below) of objects that appear more than once on the tape.

By default, **mxload** writes a map of everything reloaded to standard output. This map gives each object's Multics pathname, its date/time modified and date/time used, its size, its object type and **mxload** conversion type, and the UNIX pathname where the object is being reloaded. If desired, this map can be directed to a different file or to files in each reloaded directory, and the amount of information in the map may be adjusted.

Examples of mxload's Interactive Style

In interactive style, the most common operation is to reload some Multics subtree into a new location in a UNIX file system. Some examples:

```
mxload /dev/rmt0 '>udd>ATMOS>Dillman' /usr/dillman
```

This will reload Ms. Dillman's Multics hierarchy into a corresponding UNIX directory (which is created if it does not already exist). All the file permissions will be set to the UNIX default (using the UNIX process's **umask**(1) value), and all the files will be owned by the process running **mxload** (Ms. Dillman's, presumably). The Multics data is read from the tape mounted on the **mt0** device (as with most UNIX tape operations, the name of the "raw" device should be used). Note that in this, and all other examples, the Multics pathname **must** be enclosed in quotes, because the ">" characters are special to the UNIX shell.

```
mxload -g map1 /dev/rmt0 '>' .
```

This will reload everything on the tape into directories under the current working directory. One might do this to reload an entire tape, and then use UNIX commands to redistribute the data it contains. Permissions and ownership will still have default values. The **−g** option directs the reload map into the file called **map1**, also in the current working directory.

```
mxload -n -c uid_list.mxl /dev/rmt0 '>user_dir_dir>ATMOS' /usr3
```

This would reload all the ATMOS project home directories into the **/usr3** file system, and restore file ownership and permissions as specified by the statements in the **mxload** control file named **uid_list.mxl**. The **−n** option suppresses generation of any maps.

```
dd if=/dev/rmt0 of=tape_file bs=4680 files=50
mxload -lx tape_file '>udd>TX1' /usr1 '>udd>TX2' /usr2
```

[2] NOTE: The **−f** option is not supported in release 1.0.

This example shows how to request multiple subtree reloads on a single command line, and also illustrates how **mxload**'s input can come from a file, rather than a physical tape. The **dd**(1) command is used first to copy the contents of the tape into a file (see *Notes on Using Disk Files*, below), then **mxload** is used to reload some Multics subtrees from that file. The **−lx** options cause **mxload** to write map files in each directory reloaded (**−l**) and to include the "eXtremely verbose" (**−x**) information about all the Multics attributes.

Example of mxload's Batch Style

The batch style is more difficult to illustrate, because the heart of the matter is the contents of the control files, which are not shown here. Several sample control files are included on the installation tape; see *Control File Syntax*, below, for details of control file syntax. This example illustrates the **mxload** command syntax for batch style operation:

```
mxload -c dir_list.mxl -c uid_list.mxl -g full_map -v
```

This assumes that a list of data to be reloaded is specified by several **subtree** (or **file** or **directory**) statements in the file **dir_list.mxl**. It also assumes that **uid_list.mxl** contains a set of **owner** and **group** statements for converting Multics Person-IDs and Project-IDs to UNIX owner and group IDs for the files being reloaded. The **-g** and **-v** options are used to create a verbose map in the working directory.

This example illustrates the case where a large Multics hierarchy is being reloaded wholesale into a UNIX system, with individual person and project directories being redistributed to new homes in the UNIX file system. Typically, the **dir_list.mxl** file would be set up by making a map of the tape (with **mxmap**, or by looking at the original Multics backup_dump map), planning the new UNIX file system organization in advance, and then reloading data from all the tapes from a complete backup_dump of the Multics file system. Actually, the command line in the example would be issued many times, once for each tape to be reloaded, and each time, it would reload more data and append to the map.

Reload Name Translation

On some target systems, the maximum length or permitted character set for filenames is insufficient to represent Multics names. In these cases, characters must be translated and names shortened. The supported name types are described in the following table (these are all keywords for the **name_type** statement in an **mxload** control file; see *Option Statements*, below):

BSD	This applies to SunOS and 4.2/4.3 BSD UNIX systems. These have practical length limitations (individual names of up to 256 characters are permitted). To accommodate the deficiencies of UNIX shells in handling special characters, both single quotes (') and double quotes (") are translated to hyphens (-). Because slash (/) is used as the UNIX pathname separator, it, too, is translated to a hyphen (-). Greater-thans (>) in Multics pathnames are (implicitly) translated to slash (/) in UNIX pathnames. Null characters in Multics names (a rarity) are also translated to hyphens (-). All other characters are left unchanged in UNIX pathnames.
SysV	This applies to System V UNIX systems. All the same translations are made as for **BSD**, but an additional limitation of 14 characters maximum name length is applied.

A Multics name longer than 14 characters is truncated (first) to 14 characters. If this results in a conflict with any existing file, it is then trimmed to 12 characters, and each of the numeric suffixes **#0** to **#9** is applied, until one results in a name that does not conflict with any existing files. If that attempt is unsuccessful, another character is trimmed from the Multics name, the suffixes **#10** to **#99** are tried, and so forth.

MSDOS This applies to PC-compatible systems running MS−DOS or PC−DOS (version 2.0 or later). The restrictions here are much more stringent: an 8-character name plus a 3-character extension. The first and last components of the Multics name are used (if there is only one, the MS−DOS name has no extension). If a name duplication occurs, the numeric suffixing described above for **SysV** is done on the first name component (which is first truncated to 8 characters, then to 6, then 5, etc.). Multics names are also translated to upper case to become MS−DOS names, and all characters except letters, digits, dashes, and some others are translated to hyphens.

CMS This applies to systems running IBM VM/CMS. These restrictions are very similar to the MS−DOS restrictions, except that the extension is 8 characters instead of only 3. VM/CMS reloads must specify the **reload flat;** statement, since no directories are available.

Ordinarily, **mxload** is compiled with the appropriate value for the option for name conversion, and the **name_type** statement need never be used. If **name_type** is used on a system that cannot support the resulting type of names, the results will be surprising.

Reloading Directories

There are two important ways in which the reloading process differs from a Multics backup_load operation: what gets reloaded, and how names are translated during the reload. For the most part, these differences are invisible, but some possible problems are described below.

A Multics backup_dump tape contains some records describing directories and other records describing segments. Each segment and directory on the tape is identified by its full Multics pathname, with all parent directories back to the root of the dump identified by primary name. If the backup_dump tape is from a complete dump, the root of the dump is usually the real root directory. If, however, the tape was created by an explicit backup_dump command that specified a pathname, the root of the dump is the pathname that was specified to the command.

It is important to understand how data is stored and named on a backup_dump tape because **mxload** only understands the pathnames stored with the segments themselves. Although the Multics retrieve command can, in some circumstances, handle pathname components that are not primary names of directories (by reference to the hierarchy or by reading directory records from the tape), no such capability is provided in **mxload**. If a segment or subtree is specified by name for reloading by **mxload**, the name specified must match the name on the tape, which generally means it must consist entirely of primary names. If there is any question about pathnames on a tape, **mxmap** can be used to display them.

The other important difference between **mxload** and the Multics retrieve command is that **mxload** does not reload directories. Although there are records on the tape for Multics directories, **mxload** ignores them, and instead creates directories on the target system as needed; that is, whenever a pathname on the tape includes a directory that does not (yet) exist on the target system. Only Multics segments are actually reloaded from the tape. Neither directories nor links are reloaded from the tape.

The effect of this is that directories are created on the target system only as needed, with their names appropriately translated from the Multics pathnames. If a directory already exists on the target system, it will be used, and its attributes (permissions, ownership, etc.) will remain unchanged. If a directory does not exist, it will be created the first time it is needed for reloading a segment. When a directory is created like this, its access attributes (permissions, ownership) will be set the same as those of the file being created, not to the directory's attributes on the dump tape. The newly-created directory's access and modification times are never set explicitly, but are simply the time it is accessed or modified during the reload.

Reloading Directories: Access Control Problems

Ordinarily, this directory creation algorithm generates the desired results, but it will occasionally cause incorrect ownership or permissions to be set. These problems are serious only when (A) **mxload** is being run by super-user, and (B) **mxload** is performing translations from Multics person and project names to user and group IDs on the target system.

Because a directory's attributes are taken from the first object that causes it to be created, a directory containing segments belonging to several users may have incorrect permissions or ownership set. The same incorrect setting may occur for parent directories further up in the hierarchy; for instance, if **mxload** is reloading from the root (>) and the first item on the tape is several levels down in >udd>INCO>Jenkins, all the directories created on the target system (**udd**, **INCO**, **Jenkins**) will be created as if owned by Jenkins, rather than with the attributes they had on the Multics system. This may result in unintentional granting access to directories created during a reload; however, since UNIX access control depends on the object's permission bits alone, this will not grant excessive access to reloaded objects. The only unintentional access that may occur is such objects may be deleted. If this is a problem, the top-level UNIX directory for the reload should be protected against unintended access by manually restricting its permissions before the reload, thus cutting off access to anything below it.

Reloading Directories: Name Problems

The other problem that the directory creation algorithm causes is in name conflicts. This occurs when the target system has short or restricted names (and thus is not a problem for systems with **BSD** name translation). Because directories on the target system are used if they already exist, with no conflict handling, two different Multics directory names (such as very_long_name_1 and very_long_name_2) that translate to the same target name (such as very_long_name on systems with **SysV** name translation, which are restricted to 14-character names) will refer to the same directory on the target system. The effect of this is that segments from the two different Multics directories will be reloaded into the same directory on the target system; if the segments have the same names, the usual name conflict handling will be used to make them different.

This usually isn't a problem, because such name limitations have to be dealt with throughout the reload. Unfortunately, there is no way for **mxload** to detect automatically when this happens, although it will be evident from examining the reload map.

The underlying cause of this is that **mxload** processes backup_dump tapes one at a time, whereas Multics processes them as a set. Thus, when **mxload** starts reading a new tape, it knows nothing about what has been reloaded already, and it can't tell whether a directory name conflict is caused by an actual conflict, or simply results from the directory already having been reloaded from a previous tape.

WARNING: On UNIX and MS−DOS systems, a Multics directory named "." or ".." will have its contents loaded into the current (or parent) UNIX directory from which the **mxload** command was issued, rather than creating a new directory. This results from the special interpretation placed on those names by UNIX and MS−DOS. A Multics segment with one of those names will not be affected, as it will be treated as an ordinary name conflict and be given a new name.

Reload Map Format

The **mxload** and **mxmap** programs can generate a map showing the contents of a tape. Two options, **−v** and **−x**, control the amount of information written to the map.

The **mxmap** program always produces a map, writing it to standard output.

The **mxload** program produces a map unless requested otherwise by the **−n** command line option. By default, the map is directed to standard output. The **−g** option may be used to direct the entire map to a specific file, and the **−l** option may be used to write the map into a set of local map files named **mxload.map**, one for each directory reloaded, in the each reloaded directory.

By default, the map is produced in a short form that lists each reloaded object on a single line, beginning with a single space, containing the object's **original** Multics entry name, its length in Multics pages (4K byte units), its type (**SEG**, **DIR**, or **LINK**), its Multics date-time-contents-modified, and its Multics date-time-used. An object line will be preceded by a line giving the absolute pathname of its parent directory if the object has a different parent than the previous object in the map. The parent directory lines start in the first column, and thus begin with ">" rather than a space. Optionally an object's line may be followed by additional lines (each beginning with two or more spaces) giving details about the object's attributes.

Two options control the amount of information in the **mxload** map. The **−v** ("Verbose") option produces the following information, in addition to the default described above:

> the object's additional names on Multics;
> the full UNIX pathname into which the object was reloaded;
> the conversion technique used to reload the object (see *Data Formats and Conversion*, above)
> the UNIX permissions, owning user ID, and owning group ID;
> the object's Multics author and bit count author;
> the object's Multics bit count and length in K bytes; and
> the object's Multics ACL.

Note that **mxmap** listings do not include the UNIX pathname, the conversion technique, or the UNIX permissions, user ID, or group ID.

The **−x** ("eXtremely verbose") produces the following information in addition to that produced by a **−v** listing:

the object’s Multics date-time branch modified;
the object’s Multics safety switch;
the object’s Multics current length and records used;
the object’s Multics unique ID, in octal;
the object’s Multics ring brackets;
the object’s Multics access class, in octal, and,
in the object’s Multics ACL listing, the binary form of the extended access modes.

The access class is normally listed as "L:CCCCCC", where each character is an octal digit. L represents the security level (from 0 to 7) and CCCCCC is a six-digit representation of the Multics categories. If the access class for the object has non-zero bits outside the 21 normally used, the entire 72-bit access class is displayed in octal, following the L:CCCCCC value. If the access class is all zeros, it is not included in the listing.

The extended access modes for an object are displayed following the name in each ACL term, in parentheses. Only the significant non-zero bits of an extended access mode are shown.

USING THE STANDALONE UTILITIES

Three major standalone utilities are provided as part of the **mxload** package for handling complex Multics objects: **mxarc** for Multics archive segments, **mxmbx** for Multics mailboxes, and **mxforum** for Multics Forum meetings. All of these perform the same basic unpacking functions. A fourth, **mxascii**, is used for forcing 8−bit ASCII conversion after a file has been reloaded in 9−bit binary format.

Complex Object Conversions

Four basic operations are available for complex objects:

−t (table)	Generate a table of contents for the object.
−u (unpack)	Replace the reloaded object (file or directory) with a directory containing one file for each of the object's components (or messages, or transactions).
−r (repack)	Extract the object's contents and re-pack them into a single file (**mxmbx** and **mxforum** only).
−x (extract)	Extract contents of the object into the current directory. For archives, individual components can be extracted.

By default, **mxload** does not perform any automatic conversion of complex objects except wholly ASCII archives. It cannot do so for Forum meetings (because they may be split across multiple tapes, and therefore **mxload** can't always recognize them). For the other objects, the reason is efficiency: it is always possible to perform the conversion manually afterward, and **mxload**'s goal is to get the data off the tape rather than to process it completely. For archives and mailboxes, the default can be overridden by the **convert** statement in a control file.

The **table** operation is useful for perusing an object's contents interactively, often prior to an **extract** operation. This is probably more useful for archives than anything else.

The **unpack** operation, which replaces the reloaded object with its closest UNIX equivalent, is used primarily for large-scale conversions. Because it creates individual files for each item in the object, its output is most flexible, but least efficient. As with any operation involving a large number of files, it may be very expensive to extract 1000 messages from a large mailbox.

The **repack** operation, which creates a single file containing the object's contents, is more efficient, but may be less flexible. The single file it creates can be processed with **sed**(1) or editor macros to convert, say, a Multics mailbox's contents into a UNIX mail file.

CONTROL FILE SYNTAX

Control files determine how **mxload** loads data from the tape. Statements in the control files specify loading options and identify objects to be reloaded. When multiple control files are used, they are processed in sequence, so that a control file of static defaults (such as conversion options and ownership translations) may be combined with a control file created to list specific objects for reloading.

Defaults and Control File Order

Actually, the **mxload** defaults and the command line options for naming subtrees may be thought of as “virtual” control files, processed in combination with the explicitly specified (using **−c**) files, as follows:

mxload defaults

The control file representation of these defaults is shown below. These are processed first by **mxload**, and may be modified by options in other control files or command line options. The default options do not specify any objects to be reloaded.

Explicit Control Files (**−c** option)

Zero or more control files may be specified on the **mxload** command line. If none are specified, the defaults apply. Objects to be reloaded may either be specified here or as command line pathnames, but at least one object must be specified by a control file or a command line pathname.

Command Line Pathnames

Pairs of *Multics-Path* and *UNIX-Path* arguments may be specified on the command line. In effect, these are translated into **subtree** and **new_path** statements in a virtual control file that is processed after the defaults and the explicitly specified control files. Only entire subtrees may be reloaded this way. Because the *Multics-Path* member of the pair contains “>” characters, it **must** be enclosed in quotes on a UNIX system.

Statement Syntax

Statements consist of a keyword and zero or more operands, terminated by a semicolon. The keyword and operands are separated from each other by whitespace (space, tab, newline). Blank lines and other whitespace between statements is ignored. Comments may be interspersed anywhere whitespace is permitted using the PL/I syntax (beginning with /* and ending with */).

There are two classes of statements: *object statements* and *option statements*. An *object statement* identifies a specific object to be reloaded: a file, directory, or entire subtree. An *option statement* affects how objects are reloaded.

There are two types of option statements, distinguished by the case of the keyword in the statement. An option statement whose keyword begins with an upper case letter specifies a default for all the reloads done with this control file. An option statement with a lower case keyword specifies an option affecting only the previous object statement. All default option statements must appear before **any** object statements, and the keywords of object statements must all be lower case. An object statement may be followed by any number of option statements modifying it, though usually only a conversion type is important (in addition to the **new_path** statement, which is required).

Object Statements

There are three object statements (**subtree**, **file**, and **directory**) identifying objects to be reloaded, and one mandatory option statement (**new_path**) which must follow each object statement. Each of the three object statements must be followed by a **new_path** statement that identifies a UNIX (or other native system) pathname where the object is to be reloaded. These are illustrated below. Keywords are in **bold** and operands are in *italics*.

subtree *Multics-Directory* **;**

The **subtree** statement specifies that *Multics-Directory*, plus all the files and directories below it, be reloaded. New UNIX directories will be created (unless the reload type is **flat**, see *Option Statements*, below) corresponding to each Multics directory.

file *Multics-File-Name* **;**

The **file** statement specifies that a *Multics-File-Name* be reloaded at the designated location. This is useful for picking individual items from a tape.

directory *Multics-Pathname* **;**

The **directory** statement specifies that a *Multics-Directory* and the files it contains, but not any subordinate directories, be reloaded at the designated location. This is distinct from the **subtree** statement, and is primarily useful when reloading a complete subtree with some exceptions: the **subtree** is specified normally, followed by several **directory** statements with **conversion discard ;** statements.

new_path *UNIX-Pathname* **;**

This is properly an option statement, modifying the previous object statement. It specifies that the object be reloaded at *UNIX-Pathname*. If **new_path** is not specified, it is equivalent to **new_path . ;** (that is, the object will be reloaded into the current working directory).

If an object is matched by more than one type of statement (e.g., it is named explicitly by a **file** statement, but in a hierarchy specified by a **subtree** statement), the options for the **file** statement take precedence over any options in the **subtree** statement. A **file** statement takes precedence over a **directory** statement, which takes precedence over a **subtree** statement.

Option Statements

The following option statements are permitted. Each is shown first in its default form (that is, the form in which it appears in the virtual control file representing **mxload**'s defaults), followed by its other forms. Keywords are in **bold** (and the first in the list is always in caps, since it represents a default), and operands are in *italics*.

Convert *file-type conversion* **;**

This specifies how Multics segments of a particular *file-type* are to be converted on reloading. These conversions are described in detail above (see *Data Formats and Conversions*, above). This table identifies the valid conversions for each type.

File Type	Conversion Types
ascii	**8bit**, **discard**
nonascii	**9bit**, **8bit**, **8bit+9bit**, **discard**
ascii_archive	**8bit**, **unpack**, **discard**
nonascii_archive	**9bit**, **8bit**, **8bit+9bit**, **unpack**, **discard**
mbx	**9bit**, **unpack**, **discard**[3]
ms	**discard**, **9bit**, **unpack**
object	**discard**, **9bit**

Force_convert 8bit ;
force_convert 9bit ;
force_convert 8bit+9bit ;
force_convert discard ;

Specifies that, regardless of any per-file type conversions specified by a **Convert** (or **convert**) statement, newly created UNIX files are to be created according to this specification. This can be used, for instance, to force conversion of all components of an archive as **8bit**, even if that means discarding information from some of them.

List *list-name* **none ;**
list *list-name* **global ;**
list *list-name* **local ;**

Specifies where the lists of added names (if *list-name* is **addname**), links and link targets (if *list-name* is **link**), and access control lists (if *list-name* is **acl**) are written. The default (**none**) is not to create them at all; **global** causes them to be created in the working directory of the process running **mxload**, and **local** causes a separate list to be created in each directory reloaded, describing the contents of that directory alone. Multiple **list** statements may be specified, one for each of the three valid *list-name* values.

Person (other) (process) ;
person (other) *uid/username* **;**
person *Person-ID* **(process) ;**
person *Person-ID uid/username* **;**

Specifies how to translate the Multics "ownership" of a file to the appropriate UNIX owner attribute. The keyword **(other)** (the parentheses are part of the keyword, to distinguish it from a name) indicates conversion for any names not explicitly specified by a **person** statement with a *Person-ID* operand. The **(process)** keyword specifies that the user owning the process running **mxload** should end up as the owner of reloaded objects; by default, all objects are treated this way. Otherwise, the *uid/username* form may be used to specify a particular numeric user ID or name. Although shown here in both its default-setting and per-object form, it will be very rare for the **Person** (or **Project**) statement to be used except in a long list of translations to be used in setting defaults for large reloads. Access

[3] Actually, **8bit** and **8bit+9bit** conversion can be specified for **mbx**, **ms**, and **object** files, but such conversions are almost entirely useless.

restrictions on the target system will usually require that **mxload** be run by the super-user in order to set the owner to a specific user ID other than **(process)**.

Project (other) (process) ;
project (other) *gid/groupname* **;**
project *Project-ID* **(process) ;**
project *Project-ID gid/groupname* **;**

This is equivalent to the **person** statement, except that it relates Multics *Project-IDs* to UNIX group IDs. Access restrictions on the target system will usually require that **mxload** be run by the super-user in order to set the owning group to a specific group ID other than **(process)**, although this may be less restricted than setting the owner.

Reload hierarchical ;
reload flat ;

A **hierarchical** reload creates directories when reloading a subtree. A **flat** reload reloads all files from a subtree in the directory specified by **new_path**, "flattening" the reload and collecting all the files together. This is most useful for target systems with serious name restrictions, or lacking a hierarchical file system.

Access acl ;
access default ;

The UNIX permission bits for reloaded files are, by default, set to a value derived from the file's Multics Access Control List. If the **default** keyword is specified (instead of **acl**), then the process's UNIX default (the *umask*) is used instead.

Access_time dtu ;
access_time now ;

The default (**dtu**) specifies that the time-last-accessed of files will be set to the Multics date/time used. Otherwise, (**now**), no explicit action will be taken and the current time will be used.

Modification_time dtcm ;
modification_time now ;

The default (**dtcm**) specifies that the time-last-modified of files will be set to the Multics date/time contents modified. Otherwise, (**now**), no explicit action will be taken and the current time will be used.

Owner author ;
owner bit_count_author ;

This specifies which Multics segment attribute, author or bit count author, id used to determine the UNIX file's owner. The Multics Person-ID from the author (or bit count author) is translated according to the rules specified by the **Person** statements, and is set on the reloaded files. If no explicit owner (or group) setting is desired, which is the default, an appropriate **Person: (other) (process) ;** statement should be used. It will be rare for the **Owner** or **Group** statement to be used other than to specify a default.

Group author ;
group bit_count_author ;

This specifies how the UNIX file's owning group is set, like the **owner** statement. The

Multics Project-ID is translated to a UNIX group ID by the rules from the **Project** statements.

Dataend bitcount ;
dataend page_boundary ;

Specifies how the length of a reloaded object is determined. The default, **bitcount** uses the Multics bitcount, and should be appropriate in almost all circumstances. This statement will rarely be used.

Name_type BSD ;
name_type SysV ;
name_type MSDOS ;
name_type CMS ;

This specifies the type of name conversion to be performed. Ordinarily, this is useful only for testing, as the correct name conversion is usually compiled into the program. For a detailed description, see *Reload Name Translation*, above. This statement will rarely be used.

INSTALLATION

The distribution tape (or diskette) contains four directories:

source	Contains all the source and header files, and a UNIX **makefile** for compiling them. Note that all the names are lower-case, including **makefile**.
info	Contains a copy of the source (**troff −man**) for the manual pages and for this manual (format with **tbl | troff -ms**).
misc	Contains a handful of test cases, some of which are obsolete or uninteresting; these are primarily useful as a go/no-go test of the parser.
sun3	If you ordered a Sun-3 binary distribution, this contains a set of Sun-3 binaries for SunOS release 4.0. For other releases, just recompile.

Distribution tapes are in **tar** format, written by SunOS release 4.0. Distribution diskettes are in MS−DOS format.

UNIX Installation

On a UNIX system, installation is very simple: reload the tape, **cd** to the **src** directory, update the **makefile** appropriately, and type **make**. The default target is System V, Release 3; one of **BSD**, **SVR2**, or **GOULD** should be defined in the **makefile** to change that. On a Sun system, **mxload** will automatically be compiled correctly because the compiler defines the symbol **sun**, which implies **BSD**.

The **make** will create **mxload**, **mxmap**, **mxarc**, **mxmbx**, **mxforum**, and **mxascii**, which can then be installed in a local binary directory. No special permissions or ownership are required.

If problems occur, we expect they will be straightforward to solve. Feel free to call Oxford Systems, Inc. for assistance in porting (617-646-8619).

Non-UNIX Installation

On a non-UNIX system, it's much harder. Some special handling is already provided for names, as is the flat reload option, but those are only the tip of the iceberg. Character set conversions (i.e., EBCDIC) are not provided for, and **mxload**'s use of the C library functions may be too extensive for some systems. The C library usage is, however, much more conservative in this release than in the earlier beta-test versions.

Versions of **mxload** for MS−DOS and IBM VM/CMS have been created, though with some degree of functional impairment when compared to the UNIX version.

RELEASE NOTES

These notes apply to the December 1988 release (1.0) of **mxload***, and may change in future releases.*

Notes on Efficiency

mxload is faster, on UNIX systems at least, when its temporary files are in the same file system as the data being reloaded: the temporary files are **rename**()'d into place. The environment variable **TMP** may be set to indicate an alternative directory, as temporary files are normally placed in **/tmp**. This is true of all the programs in the **mxload** package (except **mxmap**), not just **mxload** itself.

On some systems, tape I/O is hideously slow. This is particularly true of streaming tape drives, because **mxload** has to do so much processing for the 9-bit/8-bit conversion in between reads that it usually can't keep a tape streaming. If this is a problem, it's best to copy the tape to disk first (see *Notes on Using Disk Files*, below). It may also be possible to use **dd**(1) with extremely large block sizes to achieve better buffering, but no experimentation has been done along those lines.

Notes on Error Handling

mxload is not particularly good about handling tape I/O errors. This results mostly from the general inadequacy of UNIX tape device drivers, but also from the many different (and incompatible) drivers on different systems. All **mxload** does is try to skip bad blocks, and gives up completely after more than 10 tape errors. If you have trouble reading a tape, it's probably best to use **dd**(1) to copy it to disk and then use **mxload** to process the disk image. Also, simply retrying **mxload** will often clear an error condition, then run to completion successfully, for no apparent reason.

The tape reading code is in **multtape.c**, and it can actually display quite a few different error messages. Those, however, are usually the symptom of some problem with the program itself (not being ported successfully), rather than an actual I/O error.

Notes on Conversion

Unpacking non-ASCII archives and mailboxes is quite expensive in CPU time. It is better to reload these as **9bit** and selectively unpack them later (that's why the **mxload** defaults are set up that way).

Unpacking mailboxes and Forum meetings is generally much quicker into a single large file than into many small files. Which option is more appropriate depends on how the data is to be processed once it has been reloaded.

There is no automatic conversion program for Multics multi-segment files (MSFs). For MSFs which are just stream output produced by Multics vfile_, the conversion can be accomplished easily using **cat**(1) to concatenate the component files (WARNING: beware of MSFs with more than 9 components, because the numeric component names will not sort in the correct order when the shell does starname expansion; large MSFs must have their components explicitly named in the correct order). Non-stream MSFs containing structured data may be converted by special-purpose programs written using the subroutines in **mxload** (**mxforum** is an example of such a program).

If you need to write additional special-purpose conversion programs, **mxmbx** is probably the simplest program to start from. None of it is *really* simple, though.

Notes on Using Disk Files

A Multics segment (or MSF) can be created using backup_preattach as follows:

```
backup_preattach output vfile_ dump-file-name
backup_dump hierarchy-path -debug
backup_preattach -detach
```

This will create the *dump-file-name* in a form that **mxload** can interpret. It will also create a backup map (!*shriekname*.backup.map), which must have its bit count explicitly adjusted because backup_dump leaves it open. An error file (*date-time*.ef) may also be created if any access errors occur while dumping. The -debug option disables use of privilege, which is appropriate for all unprivileged users.

Once created, the dump file can then be transferred to another system by tape or serial transfer (such as Kermit in binary mode). For either method, be sure not to introduce any unnecessary conversions, padding, etc. Once on the other system, it should be copied into a disk file which can then be processed directly by **mxload**.

If you use **dd**(1) to read a backup_dump tape onto a UNIX system, you must specify a large number for its **files** parameter, since every 128 blocks on a backup_dump tape appears as a separate file on UNIX. Since there can be around 40,000 blocks on a single 6250 BPI tape, a **files** value of 400 is probably appropriate.

In general, disk images of backup_dump tapes will be easier (and faster) for **mxload** to process than the tapes themselves, and images of backup_dump files easier still. If you're having trouble, try some of these alternative transfer techniques.

Notes on Supported Systems

The **mxload** package is written entirely in the C programming language, and can easily be converted to run on most systems with a C compiler and reasonable approximation to the C standard I/O library. Because the UNIX file system is relatively similar to the Multics file system, the **mxload** package was originally designed for use on UNIX systems. The standard distribution version runs on Sun Microsystems Sun-3 systems running SunOS release 3.2 or later; SunOS is a variant of the UNIX system derived from the Berkeley 4.2/4.3 BSD systems.

In addition to SunOS support, the distributed source for the **mxload** package includes support for use on System V UNIX systems and Microsoft MS−DOS and PC−DOS systems (version 2.0 and later). The primary differences in **mxload**'s operation on these systems is the handling of names: whereas SunOS (and all 4.2/4.3 BSD systems) allows file names long enough for any Multics file name to be converted without truncation, System V UNIX and MS−DOS have limited filename length and syntax, requiring conversion of Multics names.

Notes on Non-UNIX Versions

The MS−DOS version of **mxload** has worked in the past, though it has not been tested recently. It should be reasonably easy to compile, depending on your C compiler and linker (of course, it's mostly useful for testing with dump tape images, since not many MS−DOS systems can read ½-inch tapes). It's actually pretty useful that way for quick-and-dirty copies from Multics to MS−DOS, since it understands directory structure and file transfer programs don't. Just create a backup tape image on Multics (with backup_preattach), use something like Kermit to transfer it, and use **mxload** to unpack it.

A CMS version of **mxload** has been created and run successfully, but CMS is such a primitive operating system that many of **mxload**'s capabilities were eliminated. The standard CMS C compiler and library are particularly difficult; the compiler requires escape sequences (in the source code) for braces and other heavily-used characters, and the standard I/O library functions don't come close to meeting the ANSI standard. This is not a job for the faint of heart.