flx reference manual

rsts file system utility

Revision: April 27, 2016

Version: 2.6

Copyright

Feedback

The author would appreciate any comments, criticisms, or bug reports. Please note that this is an *unsupported* program, and any bug fixes or enhancements will be done strictly on a "time available" basis. Send feedback to:

Paul Koning
408 Joe English Road
New Boston, NH 03070
ni1d@arrl.net

Dedication and Credits

This program is dedicated to the memory of Simon S. Szeto and William J. Sconce, RSTS friends extraordinary.

Thanks to Fred Knight of Digital Equipment Corporation for providing the original inspiration (RSTSFLX for VMS); to DJ Delorie for his DOS version of GNU C; to John Wilson of DBit for Ersatz-11, PDP-11 simulator for the PC; and to Bob Supnik of Digital Equipment Corporation for his help in making FLX generally available.

1. INTRODUCTION 1

1.1 NEW FEATURES 1
1.2 COMMAND LINE FORMAT 1
1.3 FILE SPECIFICATIONS 1
1.4 CONTAINER FILES 2

2. COMMANDS 2

2.1 COMMAND SYNONYMS 3

3. SWITCHES 3

4. SOME EXAMPLES 4

5. MORE DETAILS ON CERTAIN COMMANDS: 5

5.1 GET 5
5.1.1 TREE TRANSFER MODE 6
5.2 PUT 6
5.2.1 TREE TRANSFER MODE 6
5.3 CLEAN COMMAND 7
5.4 CONTAINER FILES AND REAL DISKS 7
5.4.1 REAL DISK ACCESS 7
5.4.2 REAL DISK ACCESS IN DOS 8
5.4.3 REAL DISK ACCESS WITH OTHER OPERATING SYSTEMS 8
5.5 INITIALIZING A DISK 9
5.5.1 THE –MERGE SWITCH 9

6. WHAT'S ON THE KIT 9

1. Introduction

FLX is a program (written in C) that lets Unix or DOS systems read and write RSTS disks, or container files that contain disk data. It supports any size disk that RSTS supports (including “large” disks, those with device cluster size greater than 16), and any size file.

1.1 New features

The following features were added or changed since the last released version (V1.3):

- New commands:
  - `clean`—check and, if necessary, correct the file structure (for example after a crash)
- Command line editing and recall using the GNU “readline” library

1.2 Command line format

The FLX command line has the following general form:

```
   command [switches... | files...]
```

The command may be given as part of invoking the program (i.e., “`flx`” followed by the command line). In that case, the command line is executed and FLX exits. If no command is given, FLX prompts for a command, executes it, and prompts for another. Switches and file name arguments may be mixed in any order; unlike many Unix commands, there is no requirement to put the switches before the file names. However, if a switch has an argument, the argument must follow immediately after the switch.

Special case: if you invoke the program with its name and the “`disk`” command, the command will be executed (selecting the named container file) but FLX will not exit.

Commands and switches may be abbreviated to the shortest unambiguous abbreviation. They are case sensitive, and apart from a few switches are in lower case. There is no notion of “conflicting” switch (as DCL has) where one switch overrides a preceding switch of opposite meaning. If you include conflicting switches in a command, typically one will be ignored, but it doesn’t depend on the order. So don’t do that...

In Unix, FLX supports command line recall and command line editing similar to other GNU utilities, using the standard “getline” library. The DOS version does not yet support this (I hope to get it in someday soon).

1.3 File specifications

Most commands take one or more filespecs as arguments. These may be wildcarded. Beware of shell wildcard expansion; that generally isn’t what you want. So if you use wildcards, you’ll usually need to quote the filenames, or to let FLX prompt you for the command line. (When FLX prompts, shell wildcard expansion is *not* done.) However, with the “`put`” command the input filespecs are filespecs in the host system, not on the RSTS disk, and in that case you generally *do* want to let the shell do wildcard expansion for you.

RSTS filespecs accepted by FLX generally conform to normal RSTS rules, with a few limitations and additions. The allowed field are PPN, name, extension, protection (in that order). As is normal in RSTS, name and extension are *not* case sensitive.

PPN can be a normal RSTS PPN spec [*proj,prog*] or (*proj,prog*) or—to avoid some shell quoting—a Unix style spec /*proj*/*prog*/. If no other fields follow, the trailing / may be omitted. Note that the PPN must come first (you cannot put it at the end of the filename). PPN shorthands $! % & are accepted; # and @ are not since they have no obvious meaning when you’re not logged in to a RSTS account. For most commands, the default PPN is [1,2].

Unless otherwise stated below, the file name may not be null (but the extension may be). As in recent RSTS releases, trailing ? characters in name or extension may be replaced by a *. So "`a?????`" and "`a*`" are equivalent.

1.4 Container files

The RSTS disk or container file is specified in one of three ways:

1. If the `-disk` switch is present, the argument of that switch is the disk filespec.
2. Otherwise, if a default disk has been set with the `disk` command, that name is used.
3. Otherwise, if environment variable `RSTSDISK` is defined, its value is the disk filespec.
4. Otherwise, container file "`rsts.dsk`" in the current directory is used.

FLX uses read-only access to the disk or container file if the command only does reading (e.g., get, list), and requires read/write access only for commands that modify the disk.

Protection code is <*prot*> as usual.

2. Commands

The table below lists all the commands.

Table 1: Commands

`allocation`	Display disk block allocation information. The first and last pack cluster and the size of each free area is displayed, as well as summary data giving amount free, amount used, and size of largest free contiguous area. If `-brief` is specified, only the summary data is shown.
`clean`	Verify the correctness of the RSTS file structure, and correct any errors found. This also rebuilds the storage allocation bitmap from scratch. It can be done at any time, and must be done to re-enable writing to the disk if it is "dirty" (not dismounted correctly, typically due to a crash). Minor corrections are made without asking; anything substantial requires confirmation. There is also a "read only" check mode that verifies without changing anything.
`compress`	Prepare the RSTS disk or container file for compression, by zeroing out all unused clusters. Normally unused clusters contain whatever data was left in them, which makes file compression less effective.
`delete`	Delete RSTS files.
`disk`	Set the default RSTS disk or container name. If no argument is given, the default is cleared. Otherwise, it is set to the supplied value. The argument is interpreted in the same way as the `-d` switch. A second argument, if supplied, specifies the disk size for real disks, in the same way as the `-Size` switch. The default set with the `disk` command applies for all subsequent commands unless overridden by a `-d` or `-S` switch.
`dump`	Display an octal and ASCII dump (and, if `-wide` is specified, RAD50) of a file. If a PPN is given but the name is not, the UFD is dumped. If neither is given, you get an NFS dump of the disk. (Note that currently you cannot get a file structured dump of MFD or GFD.)
`exit`	Quit FLX (useful when it's in prompting mode).
`get`	Copy files from RSTS disk to your host system. The last argument is an output filespec. It may be a directory name, in which case you get a lot of separate output files, each consisting of the directory name plus the input filename. If `-tree` was specified, each file ends up in a host system directory underneath the one you specified, whose name is derived from the RSTS directory it came from. If the output is not a directory name, you get a single output file, containing the concatenation of all the input data. An input filespec consisting only of a PPN will cause the UFD to be copied. (To copy all files in a directory, specify `*.*` as the filename.)
`hook`	Write the boot block. First argument is the device type (default DL), case insensitive. Second argument is the file to hook (default `[0,1]init.sys`). Third argument is the file from which

	to read the bootstrap code (default whatever argument 2 is). All arguments are optional.
`identify`	Display summary information about the disk, such as pack label, flags, and size.
`initialize`	Initialize the disk as an RSTS file structured disk. The pack ID is the first argument; the second argument, if present, supplies the RDS level desired. Default is RDS 1.2. The MFD, [0,*] GFD, and [0,1] UFD are allocated with a clustersize of 16.
`list`	Show a directory listing. Default PPN is [*,*]. Default file name and extension are *.
`mkdir`	Create a directory. The PPN specified must not be wild. By default a no-user account is created, but the `-user` switch can be used to create a user account. A user account created this way has no privileges and requires no password to log in.
`protect`	Change the protection code on RSTS files, or set/clear the P (no-delete) flag.
`put`	Copy files from the host system to the RSTS disk. The last argument is a RSTS filespec. Default name and extension are *. The PPN must not be wild unless `-tree` is specified. With `-tree`, the output RSTS directory is derived from the last directory name in the input filespec, which must be numeric. The input filespecs are host system filespecs. Existing files are replaced by new files unless `-protect` is specified.
`rename`	Rename RSTS files. The new name is given as the output filespec. A PPN is not permitted. The default name and extension are *. Multiple input specs are allowed, and all are renamed according to the output spec.
`rmdir`	Delete a directory (account, user or non-user). The directory must be empty. Wild PPN is allowed.
`rts`	Set the RTS name attribute on RSTS files. The RTS name is the last argument.
`type`	Type one or more files.

2.1 Command synonyms

A number of commands have synonyms, often because the comparable command in RSTS has synonyms. The table below lists them.

Table 2: Command synonyms

`list, directory, ls`
`type, cat`
`exit, quit, bye`
`delete, rm`
`rts, runtime`
`rename, mv, move`
`rebuild, clean`
`initialize, dskint`

3. Switches

Many switches apply only to some commands, but they are recognized at all times. If you use a switch on a command where it does not apply, it has no effect. The table below lists all the switches.

Table 3: Switches

`-ascii`	Force line oriented transfer mode in `get` and `put`.
`-binary`	Force block mode transfer in `get` and `put`.
`-brief`	Brief directory listing (name.ext only, 5 across).
`-clustersize`	Output file clustersize in `put`, UFD clustersize in `mkdir`, and pack clustersize in `initialize`. May be negative for `put` and `mkdir`, in which case pack clustersize is used if supplied value is smaller.
`-confirm`	Ask about copying, deleting, renaming, etc., each file in commands that do file operations. This command applies to all operations that can act on multiple files, except

	for the `list` command.
`-contiguous`	Make the file contiguous in `put`.
`-create`	Create the container file as part of `initialize`. The argument is the container size; it may be a numeric argument or the name of a disk type (for example `RP04`). Disk type names are case-insensitive. All blocks in the created container file are zeroed.
`-disk`	Argument is the filename of the RSTS disk or container file. Valid with all commands. This switch specifies the disk or container file name for the command on which it is used only.
`-Debug`	Print various debug information while executing the command. Valid with all commands
`-end`	Specifies last block number (0-based) in `dump`. If the argument has a leading 0, it is octal; otherwise it is decimal. If this switch is omitted, the dump ends on the last block.
`-filesize`	Meaningfule only if -contiguous is specified; forces the output file in `put` to be that size. (If omitted and -contiguous is specified, the file is pre-extended to the same size as the input file.)
`-full`	Full directory listing (includes file attributes) for `list`.
`-hex`	Display data in hex rather than octal in `dump` command.
`-image`	Same as `-binary`.
`-long`	Same as `-full`.
`-merge`	Specifies merge data file in `initialize` command. See details below.
`-oattributes`	Display RMS attributes in octal in `list`.
`-protect`	Set P flag in `protect` command. In the `put` command, prevents existing files from being superseded by input files. In the `clean` command, makes it a read-only clean (all consistency checks are done but nothing is changed).
`-query`	Same as `-confirm`.
`-replace`	Let `rename` supersede an existing file, if the new name already exists.
`-size`	Same as `-filesize`
`-Size`	Specifies disk size, for real disk access in cases where the operating system does not make the size available to the program (e.g., Unix). Normally used along with a `-disk` switch specifying the disk name.
`-start`	Specifies first block number (0-base) in `dump`. If the argument has a leading 0, it is octal; otherwise it is decimal. If this switch is omitted, the dump starts at block 0.
`-summary`	Display only per-directory summary information (files and blocks) in `list`.
`-tree`	Copy directory trees (see below).
`-unprotect`	Clear P flag in `protect`.
`-user`	Create a user account (with no password required and unlimited disk and job quota) in `mkdir`.
`-verbose`	Display messages confirming what was done (e.g., copy operations completed). Valid everywhere, though some commands don't have anything additional to say.
`-wide`	Do wide dump (include RAD50) in `dump`.
`-Write`	Override normal write protection: allows writing on disk initialized as read-only, and allows deleting and modifying files that are write-protected (e.g., protection code <62>). Does not override the protection provided by the P (no-delete) flag.
`-1column`	Brief directory listing (name.ext only), one file name per line.

4. Some examples

```
get -v [0,1]*.rts [1,2]*.sav .
```

Copy all `.RTS` files from [0,1] and all `.SAV` files from [1,2], and put them in the current host system directory. The resulting file names match those of the input files.

```
put /usr/bin/ls [1,2]*.sav
```

Copy `ls` to `[1,2]LS.SAV`. Since the extension is `.SAV`, the RTS name is set to `RT11`, and the runnable bit set in the protection code. (This is obviously a rather silly example...)

```
rename [0,1]*.tmp [1,2]*.sav *.foo
```

Rename all `.TMP` files in [0,1] and all `.SAV` files in [1,2], changing their extension to `.FOO`. (Note that, while no PPN may be specified on the new filename, the inputs may be in several different directories, or indeed have wildcard directory specs.)

5. More details on certain commands:

5.1 Get

The `get` command can either concatenate files, or produce multiple output files. It concatenates if a single filespec is given as the output spec, and produces separate outputs if a directory spec is used.

The file transfer mode depends on whether concatenation is being done, on switches, and on the RMS attributes of the input files.

If concatenation is done, line mode transfer is done unless `-binary` is specified. (Note that the case of a single input file spec, without wildcards, to a single output filespec, is not considered concatenation. However, if any wildcards are present, that is concatenation even if only one file ends up matching the wildcard spec.)

If individual transfer is done, the transfer can be forced to line mode with `-ascii`, or to binary (block) mode with `-binary`. Otherwise, the properties of the input file control how that particular file is transferred:

- No RMS attributes (RSTS native format):
  - Text file—line mode
  - Others—block mode
- Copying a directory - block mode
- RMS sequential format:
  - Fixed length records, with recordsize a multiple of 512—block mode
  - Otherwise line mode
- Other RMS organizations—block mode

Line mode transfer works for RMS sequential files of any record format, and for “native RSTS” text files (files with no attributes).

A “text file” is one whose extension indicates it contains text, as opposed to binary data. The extensions that are considered “text” are:

```
txt lst map sid log lis rno doc mem bas b2s mac for ftn fth cbl
dbl com cmd bat tec ctl odl ps c h src alg
```

In line mode, an input file with no attributes is interpreted as a RSTS stream file, with cr/lf line delimiters. Lone line feeds without preceding carriage return are also treated as delimiters. In any case, the lines are written to the output file using standard Unix line delimiters (“newline”). (Under DOS, these are then translated to normal DOS line delimiters by the C I/O library, so you will get a normal DOS text file.) Any trailing nulls in the file are ignored.

In block mode, the input file is read block by block, and is copied exactly as is to the output file in binary mode. In this case, the output file size will be a multiple of 512 bytes.

5.1.1 Tree transfer mode

If `-tree` is specified, the input directory name is made part of the output spec (this is meaningful only when doing individual transfer and the output spec is a directory spec). The PPN is converted to a 6-digit string and that directory name and the filename are appended to the output spec. If necessary, the directory is created. So the command

```
   get -tree [1,2]*.sav .
```

would create (for example) a file `./001002/pip.sav` .

5.2 Put

The `put` command does transfer mode selection similar to `get`, but since there are no attributes on the input files, the rules are simpler:

The transfer can be forced to line mode with `-ascii`, or to binary (block) mode with `-binary`. Otherwise, the properties of the input file control how that particular file is transferred:

- Text file—line mode
- Others—block mode

In line mode, the input file is read line by line. At each line ending, a standard RSTS line delimiter (cr/lf) is inserted. Normally, the input file has “newline” characters (line feed) at end of line, per Unix convention. If the input file has cr/lf pairs at end of line—for example, if it was a RSTS text file copied in block mode from somewhere—then the cr/lf is kept as the line end; no additional cr is inserted in that case.

In block mode, the file is read in binary mode, and is written exactly as is to the output RSTS file. If the input file size is not a multiple of 512 bytes, the last block is filled with zero bytes (nulls) before being written.

The output file protection can be specified explicitly. If it isn't, it defaults to 60 for non-executable files, 124 for executable files. Executable files are recognized by their extension (checked against the runnable extensions of the runtime systems I know of). An executable file will always have its RTS name set. The executable bit in the protection is turned on only if the protection was defaulted.

Large files (those of size greater than 65535) have the RTS name cleared, and the executable bit in the protection code is forced off unconditionally, in accordance with standard RSTS rules.

5.2.1 Tree transfer mode

If `-tree` is specified, the output PPN is normally wild. The input file spec is then used to construct not just the output file name but the output PPN as well. The last directory name in the path is expected to be a numeric string. If the output spec is of the form [*x*,*] then the numeric (decimal) value of the directory name is taken as the programmer number. If the output spec is [*,*x*] or [*,*] then the directory name must be at least 4 digits; the low order 3 digits are the programmer number and the high order digits the project number. (In the case of [*,*x*] the programmer number is then ignored.) Examples:

`put -tree 10/foo [2,*]`	produces `[2,10]foo.`
`put -tree 10/foo [*,*]`	is an error (too few digits)
`put -tree 4010/foo [*,*]`	produces `[4,10]foo.`
`put -tree 4010/foo [*.2]`	produces `[4,2]foo.`
`put -tree bar/foo [2,*]`	is an error (directory not numeric)

If necessary, the output directory is created as part of the copy.

5.3 Clean command

The `clean` command does essentially the same thing as the corresponding RSTS feature (the one in INIT.SYS or in ONLCLN, usually accessed these days using the MOUNT/REBUILD command). It does have a few extra features:

- "Read only" rebuild, so you can see whether there are any problems in your file system without changing anything. This is done using the `-protect` switch. If you use this, you will see all the same messages as with a normal clean. In addition, anytime the program would normally ask whether you want to proceed, it will automatically supply "yes (read-only)". Don't panic, it uses "yes" so it can continue but it does not actually do any disk writes.
- Some additional consistency checking and more explanatory messages. By default clean is silent, but you can get it to tell you a bit about what it's doing by using the `-verbose` switch. And if you use `-Debug`, it will tell you more detail than you probably wanted.
- Vastly higher performance. The RSTS version has to make several passes over the file system because it uses flags in the directories to keep track of where it has been. And it doesn't use a lot of memory for the job. Yes, these days it could, but if you had to maintain such a complex piece of code in assembly language, would you want to risk breaking it?

  The clean function in FLX, on the other hand, uses separate data structures to keep track of what places it has been. And it reads entire directories into memory, so just one big read suffices for any directory (and a write, but only if it had to change something). For example, cleaning an RL02 sized file system may take only a few seconds.

The clean function will report any file structure inconsistencies or errors it finds. If read-only operation was not specified, it will also correct such errors. If the correction is minor and doesn't cause loss of data (for example, correcting non-standard values in label fields or the like) the correction is made without asking. If the correction involves possible loss of data, you will be asked whether the change should be made. If you say no, clean will usually abort. At that point you may be able to read the affected files to capture the data you were about to lose. Then again, the error may be such that you can't get to the data involved until the error is fixed. An example of this sort of problem is duplicate allocation: two files that claim to be using the same block. (Or, worse, two directories with that sort of problem.)

5.4 Container files and real disks

When manipulating RSTS files for use with a PDP-11 simulator, container files are most often used. These are simply files, as far as the host OS is concerned, but in FLX and in the simulator they are treated as a disk that has a RSTS file structure on it. The size of the container file is the size of the disk. Container files can be created as part of the `initialize` command by using the `-create` switch. The argument can be either the desired size in blocks, or the name of a disk model (e.g., `rl01` or `rk07`).

5.4.1 Real disk access

FLX is also able to access a real disk, if the host OS supports this. For example, under Unix you can specify a disk name of `/dev/rxxx`, and flx will access that device. Note that you must specify the raw device.

WARNING!

The real disk access supports both read and write access. FLX will quite happily write on whatever you point it to, if it looks enough like a RSTS file structure (or if you tell it to create one using the `initialize` command). Be careful.

5.4.1.1 Size of real disks

Unfortunately, there is no standard way for a program to find out the size of a real disk. There isn't even a common approach that works for more than one Unix. So much for portability!

Therefore, when you tell FLX to access a real disk, you may need to specify the size as well, if for your OS FLX cannot tell. This is done by specifying the size explicitly, either numerically (in blocks) or by a disk model name, exactly as in the `-create` switch. The size can be supplied using the `-Size` switch (note upper case S), the second argument of the `disk` command, or the environment variable `RSTSDISKSIZE`. For example:

```
   dir -d /dev/rrz0g -Size 10240 [1,2]
```

5.4.2 Real disk access in DOS

There is some special case handling built into FLX for real disk access under DOS under DJGPP. Unlike Unix, it is not necessary to specify the size as described above.

With DOS, you would specify the name of a disk (for example `A:`). If that is a hard disk, FLX will use absolute DOS I/O to that disk or partition. The size is taken from what DOS reports, which seems to be close but not necessarily exactly right, so you may need to specify the actual size. (An explicitly supplied size will override what FLX obtains from the OS.) If you specify the name of a floppy disk, as in the example above, then FLX will do its I/O using straight BIOS calls. In that case, the size it gets from the BIOS should be accurate and you don't need to override it.

5.4.2.1 RX50 access

With DOS, if you access a 5.25" floppy disk (using a 1.2 MB drive, i.e., high density 5.25" drive), FLX assumes this is a PDP-11 format (RX50 format) floppy rather than a standard IBM-format floppy disk, and it adjusts the BIOS disk I/O parameters accordingly. So you will be able to use RX50 floppies directly (and you will *not* be able to use regular IBM format 5.25" floppies). RX50 floppies are 400k, single sided, 10 sectors per track, 80 tracks. All sectors are interleaved 2:1. FLX takes care of all this automatically.

5.4.2.2 A note on compilers for DOS

While FLX is generally portable, and should compile with any reasonable C compiler, that is not the case for the DOS specific direct disk access code. This code, which is in module `djabsio.c`, is written specifically for the low level DOS support features in the DJGPP port of the GNU C compiler (V1.2 or thereabouts). It should not be hard to port to other compilers that give you some sort of access to DOS and BIOS *int* calls, but it certainly won't compile without change. There is also a module `borabsio.c`, which will compile under the Borland Turbo C++ compiler, but it doesn't currently support any direct disk access since none appears to be available in the Windows version of TC++ (which is what I have).

5.4.3 Real disk access with other operating systems

The real disk access works with Unix because you can do normal I/O calls to raw devices, and with DOS because there is specific code in place to handle that case. Other operating systems are not explicitly covered. Any OS that lets you specify a device name in a file open call and then do normal read and write to it will allow direct disk access. (RSTS is such an OS; I suppose you could compile FLX under RSTS, though that would be quite a strange thing to do! Maybe if you needed read/write access to RDS 0.0 disks...) However, an OS in which direct disk access requires the use of special OS services will require changes to FLX. VMS is an example: it would require the use of `$assign` and `$qiow` rather than `fopen()` and `read()` to do real disk I/O. Anyone interested in this should create a new module that contains the appropriate routines. Use `unxabsio.c` as a template. The module should be compiled to object file `absio.o`, which is then linked into FLX. Please feed such changes back to the author for inclusion in later updates.

5.5 Initializing a disk

This section discusses in more details the steps needed to initialize a file system, in particular a bootable one.

1. The first step is to use the `initialize` command. If you need a new container file, use the `-create` switch to create it. The argument to that switch specifies the disk type (which implies the size) or it can be an explicit size in blocks. The result is a non-bootable disk with the minimal RSTS file structure on it.
2. Next, create any required directories. For a bootable disk, you will as a minimum need to create [1,2], and it must be a "user" account. The `-user` switch is used for this. You will probably need to specify the clustersize, otherwise the default (pack cluster size) is used. It's usually a good idea to select a clustersize of 16 (the maximum) for [1,2] since it has to have a lot of files in it. (Note that [0,1], which is created by the `initialize` command, is always given a clustersize of 16 for that reason.)
3. Now copy over any other files you need. In particular, for a bootable disk you will need a number of files in [0,1], such as INIT.SYS, DCL.RTS, ERR.ERR, a monitor SIL, and BACKUP.TSK.
4. Once INIT.SYS is in place, you can make the new disk bootable by using the `hook` command.

5.5.1 The `-merge` switch

This switch serves a rather specialized but useful purpose. Suppose you want to create a disk that has a RSTS file system on it but also some other file system. If that other file system does not need to use the first few blocks of the disk (used by the RSTS boot block and pack label) then this can be done. An example is the CDROM file system, defined by ISO 9660 (provided the pack cluster size is small enough; a value of 64 will not work).

To create such a hybrid disk, first create the non-RSTS file system in a container file. Make this file no bigger than necessary, since the space it takes will not be available to RSTS.

Next, initialize the RSTS disk as usual, but include the `-merge` switch with the non-RSTS container file's name as argument. FLX will copy that file into the RSTS container file, starting at block 0, replacing the data in block 0 and the pack label block with the corresponding RSTS data. Any data beyond the pack label cluster is allocated to file `[0,1]merge.sys`. Note that the size of that file will not match the container file size, since neither the boot block's device cluster nor the pack label's pack cluster are included in the size. (The size for `merge.sys` is the merge data size - DCS - PCS. If that is <= 0, `merge.sys` is not created.)

If the boot block or pack label block in the merge file contain non-zero data, you will get a warning message from FLX to report this, but the initialization will proceed. It is up to you to verify whether this is in fact a problem.

6. What's on the kit

The FLX kit contains:

- Source (`.C`) and header (`.H`) files for FLX. In the ZIP format kit, these have DOS line endings; in the tar file, they have Unix line endings. Apart from that, they are the same. As noted above, there are several source files with direct disk I/O routines, all named *xxx*absio.c where *xxx* indicates the platform. Use the one that matches your platform, or pick one to use as a starting point for creating your own if needed.
- A `Makefile`, to build FLX under Unix (with GNU GCC, or probably with any other reasonable C compiler). This file may need minor changes; read the notes in the file for details. File `makefile.dos` has been edited for use under DOS, with the DJGPP version of GCC. For other environments, read the comments in either makefile for guidance on what to change.
- `FLX.DOC`, the source file for this document (Microsoft Word format), and `FLX.PDF`, a printable version of same.

- `LICENSE`, a text file containing the BSD 3-clause license that applies to FLX. Please take a look at it, especially if you plan to make changes to FLX.

Note that FLX does not work on Big Endian systems (like Sun or Apple PowerPC based machines). You can build it, but when you run it, you'll get an error message indicating that you're on the wrong kind of system.

FLX should work properly on systems with either a 16-bit or a 32-bit "int". (It does require that "short int" is 16 bits, and that "long" is 32 bits, so DEC Alpha users may run into problems.) Most testing has been done with 32 bit int.