DJGPP Utilities Reference

This is the reference manual for the miscellaneous utility programs that come with DJGPP.

bin2h

bin2h allows a developer to embed a binary file into a source file, by converting the binary data to an array of integer values.

Usage: **bin2h datafile symbol headerfile**

The resulting header file has a format like this:

```
unsigned char symbol[] = {
... data ...
};
```

djecho

This program simply echoes whatever is given to it on the command line.

```
Usage: djecho [-n] [-o file] [-s] [-a] args...
```

`-n`
: Do not emit the trailing newline after printing the text.

`-o file`
: Send output to a file.

`-s`
: Separate args with a newline, instead of the default blank. This is useful for writing several lines of output in a single invocation (which is faster than several separate commands).

`-a`
: Append the output to the file instead of overwriting it. Useful with the `-o` option.

The purpose of this program is to be able to interact directly with make's ability to pass long command lines to programs. If you use redirection, you aren't always able to handle long command lines (especially since echo is normally in `command.com`). With djecho, you can, and you can use the `-o` option to redirect output to a file (and use the file later as a response file) without using DOS's redirection.

Latest versions of ported make do not invoke `command.com` for redirection and echo commands, but djecho is still useful for related tasks, e.g. for writing several lines to a file in a single invocation. Since djecho is part of the basic DJGPP development environment, it is easy to rely on it being present when a package is built.

djtar

Usage: `djtar [-n changeFile] [-e dir] [-o dir] [-t|-x] [-i] [-v] [-p] [-.|-!.] [-d|-u|-b] tarfile`

djtar is a program that is designed to ease the problems related to extracting Unix tar files on a DOS machine. The long file names and illegal characters make regular tar programs useless. What djtar does is make some obvious name conversions automatically and give you an opportunity to provide alternate names for more difficult cases, where several filenames map to the same name in the restricted 8+3 DOS filename space.

djtar can read compressed tar files and decompress them on the fly. (This can be also achieved by piping the output of gzip to djtar, but for very large archives you can run out of free disk space, because DOS implements pipes as temporary disk files.) djtar knows about all the compression methods supported by gzip, namely, deflation (by gzip), LZW compression (by compress), LZH compression (as with compress -H available with some versions of compress), or Huffman compression (by pack).

djtar can decompress `.tar` archive files compressed with bzip2, even if several of such bzip2-compressed `*.tar.bz2` or `*.tbz` archive files have been concatenated into a single file.

djtar can decompress files compressed with zip or pkzip as well, but only if the `.zip` file contains only files compressed with either the `deflation` or `stored` (i.e., uncompressed, like with `pkzip -e0`) methods.

djtar can also read tar archives (compressed or otherwise) written on a Unix machine to a diskette as a raw device (i.e., without DOS filesystem). To use djtar with such archives, specify a name of the drive (like `A:`) as a tarfile argument.

When extracting files from compressed archives, djtar will validate them with whatever reduntant checks are available with the compression format, and print diagnostic messages whenever these checks fail. The checks and the diagnostics are generally identical to those performed by gzip. Some of the diagnostics are only printed under the verbose operation switched on by the `-v` switch. Unless you use the `-i` switch, djtar will refuse to create files and directories beyond the first tar directory entry which fails the checksum test; this usually means the file is either corrupted or not in a valid tar format.

An additional bonus of using djtar is that it can convert Unix-style text files to DOS-style text files, and vice versa. By default, it will write all files exactly as found in the archive, but the `-u` and `-d` options can change that. djtar will always convert DOS batch files to DOS-style text files, if they have been stored as UNIX-style text files. This is because `command.com` refuses to run batch files that contain UNIX-style EOLs.

djtar performs a number of file name conversions in an attempt to make the files fit into MS-DOS's restricted file names. Any file ending in `.info-n`, where n is a number, becomes `.in`. Any file ending in `.tar.gz` becomes `.tgz`. Any file ending in `.tar.bzip2`, `.tar.bz2` or `.tar.bz` becomes `.tbz` and any file ending in `.bzip2` becomes `.bz2`. Any `++` string within a file name becomes `xx`. Any leading dots are changed to underscores (but current and parent directories, `./` and `../` are left alone). Any remaining multiple dots are changed to dashes, unless the part before the dot is shorter than 3 characters and there are more than 1 dot in the filename, in which case the dot also becomes an underscore. As a result, e.g., `.foo.a.b` becomes `_foo.a-b`, and `sh.lex.c` becomes `sh_lex.c`.

While djtar is running, if it cannot successfully perform an exclusive open of the given file (it will refuse to overwrite an existing file), it will prompt you for a new name. You may type in either a complete path, a replacement file name (no directory part), or just hit return (the file is skipped).

If djtar is called as djtart, it behaves as if it were called with the `-t` switch; when called as djtarx, it behaves like djtar -x. Thus you can create 2 links to `djtar.exe` which will save you some typing.

By default, djtar just lists the contents of the archive(s) given as its arguments. If you want to extract files, you must use the `-x` option.

Options:

`-x`
: By default, djtar just lists the contents of the archive(s) given as its arguments. If you want to extract files, you must use this option, or call djtar via a link named djtarx.

`-t`
: This option causes djtar to only print the contents of the tar file, without actually creating any files. All the decompression, CRC checks etc. are still performed; thus, this option can also be used to test archive integrity. This is the default behavior of djtar, unless you call it via a link named djtarx.

`-v`

This option modifies the output format slightly to aid in debugging tar file problems. It also causes djtar to emit more verbose warning messages and print the compression method for compressed archives.

`-.`

Enable the automatic conversion of dots to underscores and dashes. This is the default.

`-!.`

Disable the conversion of dots. You get the control of file renaming when names of several files clash.

`-n changeFile`

This option allows you to specify a list of filename conversions ahead of time. Thus, you can use djtart or djtar -t to list the files, figure out the new names with your favorite editor, and supply that file to djtar. This way, you have the fullest possible control on how the extracted files will be named on your system.

The format of the `changeFile` file is like this:

```
dir/dir/dir/old.name.here dir/dir/dir/newname.hre
dir/dir/dir/old2.name.here dir/dir/dir/newname2.hre
dir/dir/dir.to.skip.here
```

The directories must be complete, not relative. The ‘‘old’’ directories must match the complete path in the tar file, and the ‘‘new’’ directories indicate where the file goes on the DOS disk. If there is no ‘‘new’’ directory specified, the ‘‘old’’ one and all its siblings will be not extracted.

`-d`

Convert all text files to DOS text format on output. This won’t usually affect binary files, but as djtar detects the file type by looking at its first 512 bytes, a small possibility of a binary file rendered useless by this conversion still exists, so you’re advised to only use this option with archives of text files.

By default, djtar writes all files exactly as it finds them.

`-u`

Convert all text files to UNIX text format on output. This won’t usually affect binary files, but as djtar detects the file type by looking at its first 512 bytes, a small possibility of a binary file rendered useless by this conversion still exists, so you’re advised to only use this option with archives of text files.

By default, djtar writes all files exactly as it finds them.

`-b`

Write all files exactly as found in the archive, with no conversions of text files. This is the default. Files written to the console (when `-p` is in effect) are always written in text mode, so this option doesn’t have any effect when used with `-p`, unless the output of djtar is redirected to a file or a pipe.

`-e string`

Only extract files whose full path names do **not** begin with string. This option can be used to skip portions of archive. If both this and `-o` options are specified, then this option has precendence. In other ways `-e` is similar to `-o` option.

`-o string`

Only extract files whose full path names begin with string. This option can be used to extract portions of archive. Files which aren’t extracted will still be shown, but with a string `[ skipped ]` appended to their names. When given the `-o` option, djtar actually checks if string is the initial substring of each filename, so you can specify incomplete file names, thus using `-o` as a poor man’s wildcard facility. You may specify multiple `-o` options to extract several different directories and files.

`-i`

Attempt to unpack damaged archives. By default, djtar will refuse to create files whose directory entries in the tar archive fail the checksum test, and any files beyond that point. This option tells djtar to ignore the checksum test and proceed anyway. Checksum failure usually means that the tar file is either corrupted or not in valid tar format, so this option is meant to be used either with the `-t` option, or in the rare cases when you *know* for sure the tar file is valid (like if you had to edit it to change the filenames).

`-p`

When given this switch, djtar will write the files to its standard output, which can be piped into another program (like a pager) or redirected to a file. This option is designed to allow you to view or extract individual files (e.g., the `README` files) without having to open the entire archive, and should be used with an appropriate `-o filename` option.

When this option is used, diagnostic messages will be directed to the standard error stream (as opposed to standard output in normal operation), so that they won't get mixed with the files' data.

dtou

Usage: `dtou [-b] [-h] [-r] [-s] [-t] [-v] [-vv] files`

Each file specified on the command line is converted from dos's CR/LF text file mode to unix's NL text file mode.

All djgpp wildcards are supported. Timestamps of the files are preserved.

dtou will pass an exit status of 0 to the calling context if all the files have been successfully processed and an exit status greater than 0 if not. In this case, the exit status is equal to the amount of unsuccessfully processed files. A file has not been successfully processed if some kind of I/O error occurred.

Options:

`-b`
: Creates a backup of the original file if the file has been modified. `.d2u` is used as backup suffix. On systems with LFN support, the backup suffix will be appended to the file name. If no LFN support is available the backup suffix will overwrite the original file suffix.

`-h`
: Displays a help text and exits.

`-r`
: Repair mode. This mode transforms MSDOS-style EOL (CR/LF) into UNIX-style EOL (LF). It ignores Cntl-Z thus it will not truncate the file. CR sequences in front of LFs are left unchanged. This mode is intended for repairing files that have erroneously been transmited in text-mode instead of binary-mode during a FTP session or with Windows versions of Netscape which have the bad habit to transfer files with such extensions as `.gz`, `.bz2`, `.tgz` as text files.

`-s`
: Strip mode. It transforms MSDOS-style EOL (CR/LF) into UNIX-style EOL (LF) and strips a CR sequence of arbitrary length from a file, if the sequence followed by a LF. CR sequences that are not followed by a LF are left unchanged. This mode is intended to repair files that have been processed with buggy ports of Unix software, which always blindly add a CR to a LF, even if there is already a CR there.

`-t`
: Timestamp. With this option the timestamp of a modified file will not be preserved. The timestamp of an unmodified file will always be preserved.

`-v`
: Verbose mode. Prints a single line showing the file name and if file processing has been successful or not. The only case that a file is considered as unsuccessfully processed is if an I/O error has occurred.

`-vv`
: Very verbose mode. Prints the file name and shows the kind of modifications that have been done to the file. All possible output looks like:

```
File: foo.c
File unchanged.
At least one CR/LF to LF transformation occurred.
Warning: At least one CR sequence striped from a LF.
Warning: At least one Cntl-Z has been found. File truncated
at line n.
Warning: At least one LF without a preceeding CR has been found.
```

Of course, not all of the above lines will appear all together. The first line showing the file name will always be printed. If the file has not been modified at all, then only the next line will be printed. If the file has been modified an appropiate combination of the lines 3 to 6 will be printed, indicating what has been modified. The above output is of some use, e.g.: if dtou reports a LF without a preceeding CR this will be a sure sign that the file is either binary or has inconsistent EOL format for some other reason. If dtou reports that a Cntl-Z (software EOF) has been found, then the file will be truncated at that line and the rest of the file will be lost.

The program is backward compatible with previous program versions if no options are given at all. In this case, an occurrence of Cntl-Z will truncate the file, MSDOS-style EOL (CR/LF) are transformed into UNIX-style EOL (LF) and CR sequence stripping will not happen at all. Also the timestamp will not be alterated and no backup of the original file will be done.

utod

Each file specified on the command line is converted from unix's NL text file mode to dos's CR/LF text file mode. All djgpp wildcards are supported. Timestamps of the files are preserved.

gxx

This program is exactly like gcc (in fact, it *calls* gcc), except that it provides the extra libraries required to link most C++ programs, including the iostreams, stdc, and libg++ libraries.

redir

DOS, in its many flavors and versions, lacks a decent I/O redirection mechanism. Sure, it's got < and > and >>, but what about error messages? Lots of people ask, ''How do you send those error messages to a file?'' Well, you use a program like redir.

redir is basically a program that manipulates the standard file descriptors by copying them, closing and opening them, etc. Once it has the file descriptors where it wants them, it runs your program, which inherits the changed descriptors. Thus, redir has nearly complete control over the input and output of your program.

It also allows you to view the exit code of the program, and the elapsed time of the program, by supplying the appropriate options on the command line.

Note that redir is built with command-line expansion and response files disabled, so as to allow the application to control that themselves. This means that you can't use those features to provide redir's options or the command name, but if you use them for the command's options, the command will do the expansion if it wants to.

The exit code of redir is 1 if it exits on its own accord, else it returns the same error code as the program it runs.

Usage: `redir` [`-i` file] [`-o` file] [`-oa` file] [`-e` file] [`-ea` file] [`-eo`] [`-oe`] [`-x`] [`-t`] command [args . . .]

`-i file`
: Redirect standard input from file

`-o file`
: Redirect standard output to file

`-oa file`
: Append standard output to file

`-e file`
: Redirect standard error to file

`-ea file`
: Append standard error to file

`-eo`
: Redirect standard error to standard output

`-oe`
: Redirect standard output to standard error

`-x`
: Print the exit code of the command after it exits. If the exit code is 0..255, it is printed as is. If it is not, the low byte (0..255) is printed in decimal and the whole value is also printed in hex.

`-t`
: Print elapsed time, either in seconds (for short runs) or hours:minutes:seconds.

Options are processed in the order they are encountered. Thus, ''-o foo -eo'' means ''redirect output to foo, then redirect errors there also'', whereas ''-eo -o foo'' means ''send errors to where output was going, then move output to foo''.

Examples:

To redirect errors to a file:

```
redir -e errors.lst command ...
```

To redirect output to a file, and errors through a pipe:

```
redir -eo -o prog.out command ... | pipe
```

djsplit

The djsplit and djmerge programs are designed to assist in transporting files across unreliable channels or small media (like floppies). djsplit takes a big file and splits it into up to 1000 smaller files. djmerge puts them back together again.

Usage: `djsplit [-t] inputfile chunksize outputbase`

Each output file is made from appending a sequence number to outputbase. For example:

```
djsplit foo.tgz 1300k footgz
```

would result in `footgz.000`, `footgz.001`, etc.

The `chunksize` parameter can be expressed as bytes (NNN), kilobytes(NNNk) or megabytes (NNNm). The number NNN *must* be an integer; for example, `1.1m` will **not** work.

By default, djsplit creates the output files `outputbase.nnn` with the same time stamp and mode bits as the original file `inputfile`. (djmerge will then recreate the file's time and modes as they originally were.) If you want the output files to have the current time and the default attribute bits, use the `-t` switch.

djmerge

The djsplit and djmerge programs are designed to assist in transporting files across unreliable channels or small media (like floppies). djsplit takes a big file and splits it into up to 1000 smaller files. djmerge puts them back together again.

Usage: `djmerge [-t] inputbase outputfile`

Each input file is made from appending a sequence number to inputbase. For example, given `footgz.000`, `footgz.001`, etc,

```
djmerge footgz foo.tgz
```

would result in `foo.tgz` being created.

By default, djmerge creates the output file `outputfile` with the same time stamp and mode bits as the first file `inputbase.000`. If you want the output file to have the current time and the default attribute bits, use the `-t` switch.

texi2ps

texi2ps is a modified version of makeinfo that quickly turns texinfo files (such as info files are generated from) into crude Postscript files, ready for printing. The program is designed to produce a crude page, so please don't complain if the printed copy looks ugly. If you want clean pages, get and install TeX and print it with that. It will produce a much cleaner print. I'm also pretty sure that texi2ps won't support 100% of the texinfo documents out there. Again, if you want it right use TeX.

Note that the input files are *texinfo* files, not TeX or LaTeX or info files. In normal djgpp distributions, the texinfo files are part of the source distributions, and the info files are in the binary distributions.

The default settings produce a pretty densely packed page - small font, small margins. You can set the margins and font even smaller to save on paper, or make them bigger for more conventional manuals.

Usage: `texi2ps` [`-f` size] [`-I` dir] [`-I`dir] [`-v`] [`-m` pts] file

`-f size`
: The font size, in points. The default is 10 points.

`-I path`
: Specifies an additional path to search for include files.

`-v`
: Print verbose messages about files read, nodes encountered, and pages generated.

`-m pts`
: Set the margin size in points. The default is 54 points, or 3/4 inches (72 points = 1 inch).

Note that the Postscript file is sent to standard output, so you must redirect the output to a file or a printer.

update

update is a fancy copy-if-needed program. Basically, it compares the two files you give it, and if they differ, copies one to the other. This is useful in makefiles where a file is generated often, but its contents changes rarely, and other files depend on it. For example, bison emits a header file each time it parses the grammar. The header rarely changes, but the grammar changes often. So, you use update to copy the header to its real name only when it changes, and everything else won't need to be recompiled each time.

Usage: `update` new_file copied_to

Example:

```
foo.c foo.h : foo.y
bison foo.y
update y_tab.h foo.h
update y_tab.c foo.c
```

Another use of update might be to validate (and replace if necessary) a local copy of some master data file:

```
.PHONY: foo.dat
foo.dat: /master/data/foo.dat
[ -f $@ ] && cmp -s $< $@ || cp -f $< $@ && touch $@
```

update is similar to the GNU-standard shell script `move-if-change` that comes with some GNU packages (e.g., GCC, binutils). The two are *not* functionally equivalent, however; in particular, `move-if-change` always **removes** the source file in the process of updating the target, whereas update leaves the source file intact.

If you need a portable replacement for update, there are several options:

- Modify the GNU `move-if-change` script to perform a ''copy-if-change'' operation.
- Unconditionally copy source to target:

  ```
  cp -f new_file copied_to && touch copied_to
  ```

- Copy source to target only if target differs from source:

  ```
  [ -f copied_to ] \
  && cmp -s new_file copied_to \
  || cp -f new_file copied_to \
  && touch copied_to
  ```

An important feature of update is that copied_to is created with the current timestamp. If you are certain that cp on your system is not aliased to cp -p or cp --preserve (to preserve timestamps), then you can omit the `&& touch copied_to` bit in the above command strings.

getconf

getconf writes the value of configuration dependent variables to standard output.

Usage: `getconf` [`-v` specification] [`-h`] [system_variable] [path_variable path]

`-v specification`
: Return values for configuration variables from the compilation environment specified by specification. Use `getconf -h` to list all supported compilation environments.

`-h`
: Write to standard error all recognized compilation environments and configuration variables. Those configuration variables requiring a path argument are listed with `[path]` after their names.

When system_variable is given, the value of system_variable is written to standard output. When path_variable is given, the value of path_variable for the pathname path is written to standard output.

Examples:

```
getconf PATH
getconf -v POSIX_V6_ILP32_OFF32 PATH
getconf NAME_MAX c:/djgpp
```

go32-v2

go32-v2 is a utility which allows you to run unstubbed COFF images.

With no command-line arguments, it prints the available physical and virtual memory, much like go32 did in v1.x. It can run unstubbed v2 COFF images, like this:

```
go32-v2 myprog
```

If you rename it to `go32.exe` and put on your PATH before the v1.x `go32.exe`, it can also run v1 COFF images, by loading the v1.x go32 and letting it do the job. With this setup, you can run v2 programs from v1.x programs, because the v1.x program will load go32-v2 (since it found it first on the PATH) which knows how to run v2 images, instead the original go32 which cannot.

If you define the environment variable GO32_V2_DEBUG, go32-v2 will provide additional debugging output when it executes.

stubify

Usage: **stubify[-v] [-g] [-stubparams=param[,param]] files**

stubify is a utility that adds a DOS exectuable stub loader to the front of a COFF image. The input program may be COFF or a stubbed `.exe`, and may be COFF with `.exe` extension. The resulting file will have `.exe` extension.

Options:

`-v`
: Verbose mode. When given this switch, stubify prints verbose output on the files processed.

`-g`
: Generate mode. When given this switch, stubify will create an image containing only the stub, which can be modified by stubedit (See stubedit) to point to a different existing program in the same directory. This is used to simulate links, so `argv[0]` can be used to change program behavior or create a small alias to the other program.

`-stubparams=param[,param…]`
: Pass param[param…] to stubedit to tune the stub’s parameters. Commas in param[,param…] are converted into spaces. See the documentation of stubedit (See stubedit) for the available parameters.

stubedit

Usage: **stubedit [-h] [-v] file.exe [field=value. . .]**

stubedit is a utility that allows you to modify the DOS exectuable stub loader parameters. These parameters allow you to set the default stack size, real mode memory transfer buffer size, the `argv0` value, and DPMI provider.

If `-v` is not specified and no fields are provided after the file name, stubedit will enter an interactive mode which displays each field and allow you to change the values individually.

Options:

`-h`
: Help mode. When given this switch, stubedit prints internal help.

`-v`
: View mode. When given this switch, stubedit will display the current contents of the stub parameters instead of prompting to modify them.

`minstack=`
: Specify the minimum amount of stack space. The value may be specified in bytes, KBytes (with a K suffix), or MBytes (with a M suffix).

`bufsize=`
: Specify the size of the convential memory transfer buffer size. The value may be specified in bytes or KBytes (with a K suffix). The value must be between 2 KByte and 63 KByte.

`runfile=`
: Specify the base name of the file to actually run. This is a string value of 8 maximum characters, and it is only used when this is a stub pointing to a different image (created by stubify -g).

`argv0=`
: Specify the string to pass as file component of `argv[0]` (maximum of 16 characters).

`dpmi=`
: Specify the name of the program to load to provide DPMI services if DPMI is not currently available (maximum of 16 characters). The stub will search for this file first in the directory of the image, then each directory in your PATH, and finally the current default directory.

exe2coff

exe2coff is a utility that strips the DOS exectuable stub loader from the front of DJGPP executable image and writes a COFF image. This would be used as the first step in replacing the stub with a different stub. For example:

```
C:\> exe2coff myprog.exe
C:\> copy /b cwsdstub.exe+myprog mynewprog.exe
```

coff2exe

coff2exe is an alias to the stubify utility (See stubify).

dxegen

Usage: **dxegen outfile.dxe _symbol infile.o [infile2.o … -lgcc -lc]**

dxegen is a utility which allows you to create files which contain dynamically loadable code (DXE). This command is retained for compatibility purposes and is a symlink to dxe3gen (See dxe3gen).

The image must be self-contained, and it cannot reference symbols from the main image. As a consequence, you cannot do I/O and some other functions directly from a DXE loaded image.

Constructors, destructors, and C

++

exceptions do not work.

There is a single entry point (procedure or data block) returned. This can be a vector of routines you have created.

`outfile.dxe` is the name you want to contain your dynamic load code.

`_symbol` is the procedure name (or data structure) you want a pointer to. You must add an initial underscore for most symbols created from C.

`input.o` is created with GCC from your source. Additional arguments on the command line (.o and .a files; or other ld options) are passed to ld to resolve references to build your code.

The floating point emulator code provided by DJGPP is stored in a DXE.

See the documentation for `_dxe_load` (See _dxe_load, , _dxe_load, libc) for details on how to load the dynamic code.

dxe3gen

Usage: **dxe3gen [-o output.dxe][options][object-files][ld-options]**

```
-o output.dxe Define the name of output DXE file
-P module.dxe Specify dependency module (cumulative)
-I import.a Create an import library for given DXE file
-Y import.a Create autoresolved import library for given DXE file
-D description Set module description string
-E prefix Export only symbols that start with <prefix> (cumulative)
-X prefix Exclude symbols that start with <prefix> (cumulative)
-U Allow unresolved symbols in DXE file
-V Verbose output (minimal output by default)
--show-dep Show dependencies for specified module
--show-exp Show symbols exported by the DXE module
--show-unres Show unresolved symbols in the DXE module
[ld-options] Any other options are passed unchanged to ld
```

dxe3gen is a utility which allows you to create files which contain dynamically loadable code (DXE). DXE is used as a synonym for ‘dynamically loadable executable module’, also called sometimes ‘dynamic modules’ or simply ‘modules’ in this document. File names provided as arguments should not contain special characters that require quoting or they may not be passed properly to ld.

There are several ways to use DXE modules. You may either load/unload modules at runtime or to link with them (or rather with so-called ‘import libraries’) at link time (e.g. statically link with dynamic libraries).

dxe3gen allows you to build either dynamic modules that don’t have unresolved symbols as well as dynamic modules that DO have unresolved symbols (and allows you to resolve them at runtime). If your DXE references symbols in your main image, you must provide it the locations for those symbols or use the dxe3res command.

Statically linking against dynamic libraries allows for a easy way to introduce dynamic modules into any program. ‘Static linking’ means that during linking stage of your program you link against certain library, called a ‘import library’ in the following, which provides all the functions present in certain DXE module. The symbols points to some very small wrappers, and when you call any of those functions, the library is automatically loaded and the call is redirected further. Also you may load/unload the library manually by calling some special functions called `dlload_MODNAME` and `dlunload_MODNAME` functions (where `MODNAME` stands for the name of your dynamic module).

The dxe3gen tool can build these import libraries, so basically you just take a existing DXE module and generate the corresponding import library for this module (see the ‘Import libraries’ section). Then you link against it... and voila! it works.

Static linkage against dynamic libraries has a drawback: you *CANNOT* have exported data inside the library (at least you can’t use it). This happens because the symbols are resolved to the wrappers and not to the actual symbols inside the module. That is, if you have a variable called `i` in a dynamic module, and you assign `i = 1` from your program which is statically linked against that module, you will actually write over the wrapper and not into the actual location where the contents of `i` are. If you really need to set/get values of some variables inside a dynamic module, write a couple of set_i and get_i *functions*.

Some modules require other modules to be already loaded when they are hit (either via `dlopen()` or a call into their corresponding import library). Don’t worry about that: just use the **-P** (dependency) option of the generator.

Another way to use DXE modules is implemented through the `dlopen()` API (See dlopen, , dlopen, libc). This API closely mimics the one found on many Unix-like systems, thus you may even build some Unix applications that use dynamic libraries without a single change.

The dynamic loader supports initialization and finalization functions inside dynamic libraries. You may use the standard C/C

++

way of doing initialization/finalization, for example (C

++

):

```
class __init
{
__init ()
{
… initialization stuff
}
~__init ()
{
… finalization stuff
}
} __dummy_object;
```

The above approach will work on any platform with a C

++

compiler. During program (or shared library) startup the constructor for the `__dummy_object` is called, and during program (or shared library) shutdown the destructor is invoked. For plain C you will have to use the GNU C extensions:

```
void __attribute__((constructor)) __init_func ()
{
… initialization stuff
}

void __attribute__((destructor)) __finit_func ()
{
… finalization stuff
}
```

Building a DXE

To build a DXE module you will use the dxe3gen tool.

Now suppose you have some library which you want to turn into a DXE module. To do this inside the makefile for your library add a rule like this:

```
mylib.dxe: one.o two.o three.o four.o
dxe3gen -o $ $^
```

That is, the **dxe3gen -o mylib.dxe one.o two.o three.o four.o** will build the `mylib.dxe` dynamically-loadable module. All public (e.g. non-static) symbols will be exported, and you may query the address for any of them with the `dlsym` API function.

Now suppose you use some functions from the C library, like `strcpy`, `strlen` and so on. In this case dxe3gen will list all those symbols and will tell you that they are 'unresolved', because these functions are not present in any of the object files you have specified on the command line. From now on you have several ways to go:

-*- You can link the DXE module with the C library by specifying **-lc** on the dxe3gen command line. This effectively will add all the unresolved symbols to your DXE module, taking them from the C library. Drawbacks:

- If your program and/or other modules use same functions, this will duplicate the code through all of them.
- Many functions won't work this way because they refer to other symbols in other libraries such as **-lgcc**; the later usually refers some symbols from `crt0.o` and you can't link with `crt0.o` (well, you can but you cannot launch crt0.o's initialization routine because it will screw many things up). For example, you can't link against **-lc** to get the `printf` routine. Most file functions and memory allocation functions are a no-no as well.
- Sometimes it could lead to a deadly-embrace situation: suppose you resolve against LIBC, build an import library and then link with it. If the functions you extracted from LIBC are made visible (exported) in the module, the corresponding import library will contain references to them. Then, the linker might consider the symbols from your library, instead of LIBC. Here comes the nastiness: until the module is loaded, all those symbols are nothing but wrappers to `dlopen`. The worst case is when `dlopen` fires but calls such a function, which again will try to load the module and so on. As a workaround, keep such names unexported by filtering exported names through **-E** option.

-*- The best way is to leave the symbols unresolved and resolve them at runtime. To tell dxe3gen to not fail on unresolved symbols, add the option **-U**. This will build a DXE module with unresolved symbols.

Before loading such a module you should provide somehow the missing symbols to the DXE loader so that it can fix up all the references to those missing functions. If it doesn't succeed, the loader will fail. The easiest way to provide symbol references is using dxe3res.

There are several ways to provide symbol references to loader. The first one is to provide a explicit pointer to every function you are going to export into dynamically-loadable modules. You can do it with a couple of handy macros:

```
#include <sys/dxe.h>

DXE_EXPORT_TABLE (exported_symbols)
DXE_EXPORT (printf)
DXE_EXPORT (strcpy)
DXE_EXPORT (strcat)
…
DXE_EXPORT_END

or:
```

```
#include <sys/dxe.h>

extern_asm(_printf);
extern_asm(_strcpy);
extern_asm(_strcat);
…
DXE_EXPORT_TABLE (exported_symbols)
DXE_EXPORT_ASM (_printf)
DXE_EXPORT_ASM (_strcpy)
DXE_EXPORT_ASM (_strcat)
…
DXE_EXPORT_END
```

Both the above sequences are equivalent. As you can see, `DXE_EXPORT_ASM` macro requires its parameter to be previously declared with "extern_asm". Now you should pass this export table to the dynamic linker:

```
dlregsym (exported_symbols);
```

The loader now knows these symbols so if any loaded module has references to them, it knows how to resolve them. You may call `dlregsym` as much as you want, the symbols are accumulating in a internal table of the dynamic loader. Also you may unregister symbols with `dlunregsym` function.

Another way to resolve symbols is to make all exported symbols in one module global, thus they get added to the global symbol table and when a new module with unresolved symbols is loaded, these shared libraries are searched for the corresponding exported symbol. This allows you to use symbols from one module as regular 'extern's in another module. A simple example:

```
---------- module A
void something ()
{
…
}
---------- module B
extern void something ()
void something_else ()
{
something ();
}
```

When you link module B, you use the **-U** switch to suppress the warning about symbol 'something' being unresolved. Now from your program you first load the module A:

```
dlopen ("moduleA.dxe", RTLD_GLOBAL);
```

then load the module B:

```
dlopen ("moduleB.dxe", 0);
```

The references are now resolved. Note that during first `dlopen` call the `RTLD_GLOBAL` flag is specifed so that all exported symbols become visible to other modules.

The third and most powerful way is to use the dxe3res tool (See dxe3res). It combines the previous methods and requires minimal interaction from the user, as the table is built automatically. To be more specific, dxe3res outputs a valid C source file, which must be compiled and linked with your application:

```
dxe3res -o table.c moduleA.dxe moduleB.dxe
```

Also sometimes you may want to build a DXE module from a ready library. For example, you can build DXE modules of `libjpeg.a`, `libpng.a` and `libz.a` without recompiling the libraries. For this you should specify a special option to the linker: **--whole-archive**(this is a GNU ld option, it is just passed to ld by dxe3gen):

```
dxe3gen -o z.dxe -Y libz_i.a --whole-archive -U libz.a \
-D "Zlib compression library"

dxe3gen -o png.dxe -Y libpng_i.a --whole-archive -U libpng.a \
-D "Portable Network Graphics (PNG) Reference Library"

dxe3gen -o jpeg.dxe -Y libjpeg_i.a --whole-archive -U libjpeg.a \
```

```
-D "The Independent JPEG Group's JPEG software"
```

You should call all import libraries `libsomething_i.a`, to avoid confusion.

Import libraries

Import libraries are a special kind of libraries (`.a`) which contains a number of very small wrappers (just long jumps to the actual functions inside the dynamic module and two functions: `dlload_MODNAME` and `dlunload_MODNAME`.

Initially all jumps are directed to `dlload_MODNAME`, thus when you call any function from a dynamic module it initially arrives to `dlload_MODNAME` function. Then the module is loaded, all jumps are directed to the respective locations inside the dynamic module, and the last call is restarted.

As you may understand (and mentioned above), this approach does not work with variables. This is one of the drawbacks of this method.

To build a import library you should use the **-Y** switch on the dxe3gen command line. The import library can be built from a ready DXE module as well as 'on-the-fly' during DXE generation process. For example, you can do:

```
dxe3gen -Y libmy.a -o my.dxe one.o two.o three.o
```

as well as

```
dxe3gen -o my.dxe one.o two.o three.o
dxe3gen -Y libmy.a my.dxe
```

Both above sequences are equivalent. The **-I** switch does the same thing as the **-Y** option and is used if you do not want to have symbols automatically resolved (when they are provided by another DXE, for example).

Contacts

This library was designed specifically for the Crystal Space project (see http://crystal.sourceforge.net), but it is very general and useful in other environments as well.

Daniel Borca, <dborca@yahoo.com> Andrew Zabolotny, <bit@eltech.ru>

dxe3res

Usage: **dxe3res -o linkage.c moduleA.dxe moduleB.dxe…**

The dxe3res tool takes one or more DXE modules and creates a C source file which will transparently load the DXE as if it were linked into your image. The C source file must be compiled and linked with your application.

```
dxe3res -o table.c moduleA.dxe moduleB.dxe
gcc -c -O2 table.c
gcc mymain.o table.o
```

symify

Usage: **symify [-o output.dmp] [-i input.dmp] progname.exe**

symify is a utility that adds procedure names and source line numbers to the call frame traceback displayed on the screen when a program aborts. The default behavior is to add the extra information to the screen, but input or output from files can be specified with the -o and -i switches. symify can only interpret debug information in COFF format. If bfdsymify is installed symify will pass the arguments to bfdsymify, which can handle additional debug formats. bfdsymify is included in the gdb (GNU debugger) package.

The call frame traceback EIPs are displayed in hexadecimal address values. These values show the addresses where one procedure calls another and has pushed its return information onto the stack. The values at the top are the most recently called procedures. symify reads the debug information from the executable you specify on the command line (with `.exe` extension) and puts it after the hexadecimal numbers. If the image is stripped there is no additional information which can be added. If you compiled the modules with the -g switch to gcc you can also display source line numbers (without -g you will only get procedure names and offsets into those procedures).

Options:

`-o output-file`
: Output to the specified file output-file. symify copies the traceback plus symbolic names and puts it into the file name you specify after the -o option. This allows you to save it (for sending to the author, or examination while fixing the bug). If -o is specified the screen buffer is not changed.

`-i input-file`
: Input from the specified file input-file. symify reads the stack trace from the file name you specify after the -i option. This file could be created by `redir` (See redir) or redirecting handle 2 to a file.

Known compatibility problem (Windows NT, Windows 2000, Windows XP):

Sometimes `symify` cannot read the screen buffer to get the traceback information. In that case you must copy the screen buffer manually and save it to a file, and use the -i option.

edebug32

Usage: **edebug32 debug-image.exe[args to debug-image]**

edebug32 is a simple debugger for DJGPP images. See the internal help (**h** at the command prompt) for more information on usage. edebug32 is an expert's debugger and does not support newer debugging formats. debug32 can only interpret debug information in COFF format, which can be generated at compile time using the -gcoff option to gcc. It is recommended that new users try one of the other more powerful or easy-to-use debuggers such as gdb.

fsdb

Usage: **fsdb [-p path] [-d] [-s setup] debug-image.exe[args]**

fsdb is a full screen debugger for DJGPP images. See the internal help (press F1 key) for more information on usage. fsdb is not actively maintained and does not support newer debugging formats. fsdb can only interpret debug information in COFF format, which can be generated at compile time using the -gcoff option to gcc. It is recommended that new users try one of the newer debuggers such as rhgdb.

Options:

`-p source-path`
: Specify source path as source-path to find source files used to build this image. The source path can also be set as an environment variable FSDBPATH, but the command line option will override the environment variable.

`-d`
: Dual monitors. Enable dual monitor display using monochrome graphics card. Useful when debugging graphics images but requires special hardware. Dual monitors can also be requested by setting the environment variable FSDBDUAL.

`-s setup-file`
: Specify setup file as setup-file. The default is `fsdb.dsk` in the current directory.