LINUX

Release 1.1.45

Using DLL Tools With Linux

Version 2.16

Written by David Engel, on or about 12 Dec 1992

Rewritten by Eric Youngdale (ericy@cais.com)
to cover DLL, 30 Jan 1993

1. Introduction

The most difficult task of building a jump table library for Linux has been the handling of global data. Since data is referenced directly, its location must not change from version to version. Historically, this has been accomplished by manually identifying all global data and moving it to separate files. Very messy! Additionally, the resulting source changes are very Linux-specific and, as such, are unlikely to be integrated back into the standard distributions. This would make keeping the Linux sources for a library in sync with the standard sources a very daunting task. This is the reason there haven't been jump table versions of the X libs. That is, until now!

The programs in this kit greatly simplify the generation of a jump table lib. In fact, after a little initial setup, the entire process can be easily run from a Makefile without any further intervention. These programs were made possible by an idea from Eric Youngdale (ericy@cais.com), and the tools have undergone a lot of refinement since then. The original tools were primarily designed to deal with just a regular jump table library, but it became clear that there was a need for some form of dynamic linking, and this was incorporated as well.

This file contains a step by step description of how to build a DLL jump table library for Linux. In case of trouble, see the troubleshooting section. Within this document bordered displays highlight important points. Any line begining with a **%** character should be executed by the user and the rest is feedback generated by the respective programs.

2. Preparation

First of all you need to configure the tools a little bit, and this is done by editing the Makefile in the tools directory. There are a couple of defines that you might wish to add:

-DLONG_FILENAMES — Use this if you will be building the library on a filessytem with a filename length limit >> 14. Extfs or ext2 fit into this category. DO NOT DEFINE THIS IF YOU ARE GOING TO USE THE TOOLS ON A LIBRARY THAT IS STORED ON A MINIX FILESYSTEM. The key directory for which it matters what filesystem that you are going to be using is the one pointed to by the JUMP_DIR environment variable. This does not need to be located anywhere near the actual library source, so in principle the library could be on a minix filesystem and the JUMP_DIR could be an ext2 filesystem, and you could legitimately define this before you build.

-DIMPLIED_IMPORT — (THIS SWITCH DOES NOT WORK WELL. PLEASE CONSTRUCT THE jump.import FILE ACCORDING TO THE INSTRUCTIONS LATER ON IN THIS DOCUMENT. IT MAY BE FIXED AT A LATER DATE). Use this define if you do not wish to put together a jump.import file. This file would contain a list of all variables that we are importing from other sharable libraries, and the usual trick is to simply cat together jump.vars files from those libraries to get jump.import. This can be a little bit of a nuisance, so the IMPLIED_IMPORT switch basically tells the tools that all symbols that are *not* in jump.vars, jump.funcs or jump.ignore are being imported from other sharable libraries. Unfortunately this means that the first time you build the library (before you have these files) the `jumpas` program has to work harder rewriting assembly code, and if there is a bug in the `jumpas` assembly code rewriting, then you are more likely to encounter it with IMPLIED_IMPORT. Once you have the jump.* files, there should not be that much difference between using this switch and not using it. The advantage of using this switch is obviously that you do not need to put together the jump.import file.

There is no free lunch with the -DIMPLIED_IMPORT switch, however. The `jumpas` program has no way of knowing if an imported symbol is a function or a variable. We are only interested in variables, but sometimes functions are referenced in such a way that it is impossible for `jumpas` to tell whether it is looking at a function or a variable. The only way that `jumpas` can resolve this is to assume that it must be a variable, but this means that any functions that accidentally get identified will be referenced via a __GOT__ variable (which does not exist for functions). Thus functions like this need to be added to jump.ignore. I am not really convinced that this is any easier, so for the time being I recommend that people generate the jump.import file manually.

Next you compile the tools, and the Makefile is set up so that you can simply type `make` and all of the programs will be properly compiled and linked. To install, type `make install` as root, and all of the files will be moved to the correct locations. If you wish to install manually, just examine the install target in the Makefile.

Now you can turn your attention to the library that you wish to build. You can start by building a static version of the library (it's usually a good idea to start with a working version :). Save a copy of the Makefile. It's probably possible to have a single Makefile control the building of both static and jump table versions, but I haven't addressed that problem yet.

Make a subdirectory (I usually call it jump) to hold the intermediate files. Add a variable, called JUMP_DIR, to your environment which contains the name of the jump directory. Also, if your library is spread across several subdirectories and built with recursive makes, the JUMP_DIR variable must be an absolute path. If you are using GNU `make`, you can sometimes add JUMP_DIR to your top-level Makefile and make will usually automatically export it to the environment for you. This sometimes does not work, so you should be careful when you try this with a new library for the first time. It is always much safer to set the environment variables in a script which then calls make.

If you fail to set JUMP_DIR, then the current working directory is used to store the various files that are generated, and the tools generate warning messages.

Add another environment variable, called JUMP_LIB, to contain the basename of your target library. Eg., if you are building libedit.a, JUMP_LIB should contain "libedit". Again, if you are using GNU `make`, you can simply add JUMP_LIB to your top-level Makefile. If you are combining multiple libraries to be combined into a single shared image, you must make sure JUMP_LIB is set individually for each library.

If you are building multiple libraries, or you are combining several libraries into one sharable image, you will need to have a separate jump directory for each sharable image that you are building. For example, let us say that are building libX11, libXold and libXt and you plan to combine libXt and libXold into one sharable image, and you wish to generate a separate sharable image from libX11. You can create two jump directories, call them jump-X and jump-Xt, and then set the JUMP_DIR environment variable to point to either jump-X or jump-Xt. Since you are putting libXold in with libXt, you set JUMP_DIR to point to jump-Xt while you are building libXold. In all cases, the JUMP_LIB environment variable should be just the name of the library you are in the process of building.

Modify your Makefile(s) so `gcc` is invoked with the `-B/usr/bin/jump` option. Adding a "CC = gcc -B/usr/bin/jump" should be sufficient. However, if you are using recursive Makefiles, you'll need to make sure it isn't overridden by lower level Makefiles. Note that in previous versions of the DLL tools, a trailing '/' was required; it is now required that there be NO trailing slash, or else `gcc` will not be able to find `jumpas`.

If you are building X libraries (or anything where you use `imake` to generate a Makefile for that matter), you can add the line:

CCOPTIONS = -B/usr/bin/jump

to the Imakefile. Similarily, in a regular Makefile, you can add something like:

CC = gcc -B/usr/bin/jump

or

CFLAGS = -B/usr/bin/jump

which will do the trick. Remember to make sure certain that this is not overridden by lower level Makefiles.

3. Generating The DLL Descriptor Files - New Libraries

To build a DLL library you need to have a list of the functions and variables that this library is going to export, and a list of the global variables that this library imports from other sharable libraries. These are used to generate the jump table, to make sure that global variables remain at the same address from one version to the next of your library, and to generate the correct symbols and pointers that are used for dynamic linking. If you already have these files (jump.vars, jump.funcs, jump.import and jump.ignore), then you may go on to the next section of this manual.

If you have never built this library before, you can create these data files by simply building your library. This should create a file called jump.log in the directory pointed to by JUMP_DIR. Whenever you compile with `-B/usr/bin/jump`, any global symbols that are not listed in the files jump.vars, jump.funcs, or jump.ignore will have their names written to jump.log. Since this is the first compile, all global symbols that are defined in the library should appear in this file.

There will be one line for each symbol. The first field will be all zeroes. More on that later. The second field is the symbol type: T=function, K=constant (in text section), C=common area, D=regular data. The third field is the name of the symbol. The fourth field is the library the symbol belongs to. (Do a global substitution of "libxyzzy" in the fourth column to the name of your library.) And the fifth field is the module the symbol was defined in.
Now you're ready to create the symbol files. `cd` to the JUMP_DIR directory and run the following commands:

```
% getvars
% getfuncs
% rm jump.log
```

The programs `getvars` and `getfuncs` will check if the variable JUMP_DIR is set and create two files in the directory pointed to by it. The file jump.funcs contains the functions to be exported from the library and jump.vars contains the variables to be exported from the library. Using your favorite editor, delete the lines from jump.funcs and jump.vars for any symbols that you do NOT want to export and add them to a file called jump.ignore.

A fourth file should be created if you are building a DLL library, and this file is called jump.import. This is a list of all symbols imported from other sharable libraries. The libc library does not import any symbols since it is self contained. The X libraries do import symbols from libc, so when you build the X libraries, all variables imported from libc into the X libraries should appear in jump.import. The easiest way to generate this file is to simply copy the jump.vars files from the libraries that your sharable library will be linked to and put this information into jump.import. If your library is linked to more than one other sharable library, you should obtain the jump.vars files from *both* of these other libraries, and these should be `cat`'ed together and used for jump.import.

If you do not have ready access to the jump.* files from the libraries that you are linking to, you can use the following commands to generate the appropriate data for a jump.import file (in this case, assuming that you will be linking to libc and to libX11).

```
% nm --no-cplus -o /usr/lib/libc.sa | grep __GOT_ | sed 's/ __GOT__/ _/' > jump.import
% nm --no-cplus -o /usr/lib/libX11.sa | grep __GOT_ | sed 's/ __GOT__/ _/' >> jump.import
```

The jump.import file that is generated by this procedure will list each symbol with a variable type of "A". Do not worry about this, it is OK. In this file the only important field is the symbol name.

Moving the symbols that you don't want to export to jump.ignore will prevent them from showing up again in jump.log whenever you recompile. You should periodically check jump.log to see if any unaccounted for symbols are detected. Sometimes to build a library, a short program must be compiled and run to generate a header file of some kind or another. If this program is compiled with `-B/usr/bin/jump` then all of the global symbols from this program will show up in jump.log, and you should place all of these symbols in jump.ignore.

4. Generating The DLL Descriptor Files - Old Libraries

Let us say that you have previously generated a DLL library and you have a new version of the source code. This section describes how to generate a binary compatible version of the library with the new source code.

You start in much the same way as when generating a DLL the first time through. However there are two scenarios.

- *You have access to jump.ignore and jump.funcs files.*
  If you have the jump.funcs and jump.ignore already set up from a previous build, then when you make the library the file jump.log will contain any new global variables and functions that have appeared since the previous version. You should divide up the symbols as before, but they should be added to the end of the existing jump.funcs and jump.vars. If you change the order of any of the old entries you will not be able to generate a new binary compatible version. If you need to remove a symbol from jump.funcs or jump.vars (because that symbol is no longer provided by the library) you should be careful because this will mean that old programs that used these symbols will not be binary compatible. You can add entries with the variable/function name __DUMMY__ to hold the place if you do remove a function or a variable from jump.vars or jump.funcs. If you have an empty slot in jump.funcs you need to fill, you can also add entries with a data type of 'U' to reserve a spot in the jump table - this will insert a call to a special function that will print an error message and then call exit().
- *Attempt to build a compatable DLL library without any jump.* files.*
  It is possible if you have the stub of an old DLL library and have obtained new sources to the library, to build a compatable drop in for the library without any knowledge of the jump.* files. The process is the same as the one described above, but the stub and the jump.log file produced from a first pass compile are compared for new and/or lost variables and functions. See *Appendix A6* for an example.

5. Compiling The Library - 2nd Pass

`cd` back to your main directory and rebuild your library (i.e. recompile everything). You may need to do a `make clean` or something like it to accomplish this. The JUMP_DIR directory should now contain several (maybe many more) new files. These new files contain the source that was diverted from the normal compile path.

These new files are of the form _GVAR_nnnnn.s and _GCMN_nnnnn.s. These contain the assembly code required to define all of the global variables that you are exporting from this library. (See the appendix on `jumpas` for more details about why these files are created and what they contain). If you are not using the LONG_FILENAMES option, it is especially important that you not change the order of any of the lines in jump.vars after you have begun to compile the library. If you must change the order of the entries in this file (and you are not using LONG_FILENAMES), you may have to recompile the library before you will be able to succesfully build the sharable image.

The next step is to determine how much space to allocate for each global variable. The general idea is that we need to alter the first field for each entry in jump.vars to correspond to the amount of space we wish to reserve for that variable. It would be rather tedious to do this by hand, so the `getsize` utility has been provided to automate this procedure. To run `getsize`, `cd` to the directory pointed to by the JUMP_DIR environment variable, and use the command:

```
% getsize > jump.vars-new
```

`getsize` will essentially read all of the data files, and generate a new copy of jump.vars where the first field has been filled in with the actual size of all of the variables.

In principle this can be used as a replacement for jump.vars. However, you should add extra space to some variables to allow for future expansion. This is especially critical for variables that are structures - it would be pointless to pad variables that are simple integers. In general, it is a good idea to make sure that each variable is allocated a multiple of 4 bytes for storage - this tends to improve performance.

The main advantage of the `getsize` utility is that it shows you how large the variables already are (which comes in handy if there are some large structs lying around, and you are not too swift with hex arithmetic). `getsize` also warns you if a variable has grown to a point where it is larger than the allocation specified in jump.vars which could happen when you upgrade to a new version of a library. Typically, you could start from this output and add space for those variables that need it. Note that `getsize` shows the actual allocation, and does not reflect any padding that is already specified in jump.vars.

6. Building Stub Library And Image

OK, now you are nearly done. If everything has worked correctly, you just need to do a few more things before you will be done.

Now choose values for the following parameters:

☞ Load address: the location where the shared image will be loaded into each process' address space. It must be >= 0x60000000 and be on a page boundary. Also the end of the image must be < 0xc0000000 (the user stack starts at address 0xbfffffff and grows downward) and the image must not overlap with any other shared images that might be used by the same process. Currently part of this range is reserved for other libraries (see `Table 1` below). If you are going to distribute a library, it would be a very wise idea to register the virtual address space so that no-one else uses it for something else. The load address is specified with the `-a` switch to `mkimage` and `mkstubs`.

☞ Starting address for the data segment: If not specified, the global variables will be stored in the text segment after the jump table (this does not create problems because code pages for sharable libraries are not marked read-only). If this is specified, you can specify the actual starting address where the data segment should start. I added this switch mainly because I wanted to generate a version of libc that was binary compatible with the existing libc, and this seemed like the best way of doing it. Unless you are trying to maintain binary compatibility with an older library, it might be best to avoid this switch. The address of the data segment is specified by the `-d` switch to `mkimage` and `mkstubs`.

☞ Image name: the name of shared image file (the one that goes in /lib). Don't forget to leave room for the ".so.X.Y" part. That means you only get 7 characters to work with because of the 14 character limit in the Minix filesystem. The image name is specified by the `-l` switch to `mkimage` and `mkstubs`.

☞ Image version: the version number for the shared image. It must be two decimal numbers, major and minor, separated by a period. This can be followed by an optional patchlevel which can be from 0 to 99. Valid version numbers look like: 4.3, 4.3.1, 3.0, 3.0.3, etc. The version number is specified by the `-v` switch to `mkimage` and `mkstubs`.

☞ Jump table size: the amount of room to reserve for the jump table. It must be > 0 and be a multiple of the

Library	Start Addr.	Stop Addr.	Maintainer
libc.so	0x60000000	0x601fffff	Hongjiu Lu
libX11.so libXt.so	0x60200000 0x602a0000	0x6029ffff 0x602fffff	Orest Zborowski Orest Zborowski
libXaw.so	0x60300000	0x603fffff	Orest Zorbowski
librl.so libgr.so libf2c.so	0x60400000 0x60500000 0x60600000	0x604fffff 0x605fffff 0x606fffff	Rob Hooft Rob Hooft Rob Hooft
libXpm.so	0x60700000	0x607fffff	Mitchum DSouza
libnsl.so	0x60800000	0x608fffff	Peter Eriksson
libolgx.so libxview.so libsspkg.so libUIT.so	0x60880000 0x608b0000 0x609c0000 0x60a00000	0x608affff 0x609bffff 0x609fffff 0x60afffff	Kenneth Osterberg Kenneth Osterberg Kenneth Osterberg Kenneth Osterberg
libPEX.so	0x60b00000	0x60bfffff	Orest Zorbowski
libtcl.so libtk.so	0x60c00000 0x60d00000	0x60cfffff 0x60dfffff	Jay Carlson Jay Carlson
libWc.so libXp.so	0x60e00000 0x60e00000	0x60efffff 0x60efffff	Niku Kaitaniemi Niku Kaitaniemi
libIV.so libUnidraw.so	0x61000000 0x61400000	0x613fffff 0x617fffff	Peter MacDonald Peter MacDonald
libXm.so	0x62000000	0x621fffff	Metro Link, Inc.
libsrgp.so libsuit.so	0x62200000 0x62215000	0x62214fff 0x622fffff	Rob Kooper Rob Kooper
libOI.so libOIrg.a	0x62300000 0x62300000	0x629fffff 0x629fffff	Warner Losh Warner Losh
libld.so	0x62a00000	0x62ffffff	David Engel
libarma.so libkarmaX11.so libkarmaXt.so libkarmagraphics.so libkarmawidgets.so libkarmaxview.so	0x63000000 0x63200000 0x63240000 0x63260000 0x632e0000 0x63360000	0x631fffff 0x6323ffff 0x6325ffff 0x632dffff 0x6335ffff 0x6337ffff	Richard Gooch Richard Gooch Richard Gooch Richard Gooch Richard Gooch Richard Gooch
libwxwin.so	0x63400000	0x634fffff	Andreas Mengel
libandrew.so	0x63500000	0x637fffff	Michael Oreilly
libUil.so libUil.so	0x63800000 0x63900000	0x638fffff 0x639fffff	Harald Koenig Harald Koenig
libBLT	0x63a00000	0x63a3ffff	David Engel
libvga	0x63b00000	0x63bfffff	Harm Hanemaaijer
libmh	0x63c00000	0x63cfffff	Brandon Allbery
libisode	0x63d00000	0x641fffff	Mike Jagdis
libkhoros	0x64200000	0x647fffff	Wolfram Gloger
libitcl	0x6a340000	0x6a37ffff	David Engel
libm3*	0x64800000	0x65ffffff	MIchael R. Dagenais
libtermnet.so	0x66000000	0x661fffff	Bill C. Riemers
libgnat,libpthread	0x66200000	0x665fffff	Rolf Ebert

Table 1

page size. Make sure to leave enough room for future growth (each jump table entry takes 8 bytes). The jump table size is specified by the `-j` switch to `mkimage` and `mkstubs`.

☞ The size of the Global Offset Table (or GOT): This is only used when building a DLL, and is used in the dynamic linking libraries. It must be used if you want your library to be dynamicaly linkable - an ordinary jump-table library will be generated if you fail to supply this parameter. The size of the GOT is specified by the `-g` switch to `mkimage` and `mkstubs`.

You're now ready to build the shared image and stub libraries. The parameters listed above correspond to the `-a -d -l -v -j` and `-g` switches for the `mkimage` and `mkstubs` programs.

Once you have selected values for the above parameters, you are ready to run `mkimage`. The remaining parameters for `mkimage` are the object files to be linked together, as well as any other linker directives (such as -ltermcap and -lc) required to link the image. Note: you should begin the object file list with `--` to keep `getopt` (used by `mkimage`) from rearranging the parameters.

The `mkimage` and `mkstubs` programs must be passed the same options for them to work correctly. There is a file jump.params which contains the parameters used the last time around, and both programs check the parameters passed on the command line with the parameters in the file. If there are any differences, the programs print a message and die. The `-f` switch forces the tools to accept the new parameters as the "true" values, and these are written back to the file. This is useful if you decide to change version numbers on/or start addresses of the data segment.

The remaining parameters for `mkstubs` are the names of all the stub libraries to create. Eg. to create libedit.sa, you should use libedit. Currently, the symbols are grouped together the same way they are in the static libs. You should always run `mkimage` before you run `mkstubs` for reasons that are about to become clear. You may also specify other options to `ld` after the `--` switch, such as path names to X libraries (-L/usr/X386/lib) etc.

There is one further wrinkle that you need to be aware of. If you link your sharable library to libc, and a user links their program to your library, it is possible (especially if the program is quite simple), that there will be no explicit need for libc in your program and thus it is theoretically possible that the user's program will need libc (due to your sharable library), but it will not have instructions to explicitly load libc. These concerns are addressed by a form of handshaking between `mkimage` and `mkstubs` - `mkimage` determines what other sharable libraries must be loaded for your library to work. This list is printed out by `mkimage`, and it is also stored in the file jump.undefs which is read by `mkstubs` whenever it is run. `mkstubs` will automatically insert the appropriate symbol references in the stub libraries to force the linker to add the instructions to load all of the sharable libraries that are required by your library

There is a hack (for lack of a better word) that allows you to override the external dependencies on a case by case basis. Basically you specify something like:

```
00000080 K _version_string     libedit      version           libbar_4:libc_4
```

In this example, it will override the external dependencies for the file version.o in libedit, and instead substitute an undefined reference that will effectively force the libraries libbar.so.4 and libc.so.4 to be linked in. Generally you will not need to make use of this feature, but there are special cases every so often where individual object files within a particular shared library are basically self contained and may not require the other shared libraries that would otherwise be required. The point of having this flexibility is that it may be easier to add something at this level rather than be forced to make custom patches to a Makefile somewhere.

There is one further switch to `mkstubs` - the `-u` switch can be used to force extra, undefined symbols into the stub libraries. In general this is no longer required since the loading of other sharable libraries is now automatic, but the switch is still there in case you find a need for it.
An example of proper usage would be

```
% mkstubs -l /lib/libc -v 4.3 -a 0x60000000 -d 0x60090000 -g 4096 -j 0x4000 \
        -- libc libcurses libgcc libtermcap libdbm libgmon

% mkimage -l /lib/libc -v 4.3 -a 0x60000000 -d 0x60090000 -j 0x4000  -g 4096 \
        -- compat/libcompat.a libtermcap.a libdbm.a libcurses.a libgmon.a \
        gcc/libgcc.a libc.a
```

The `mkstubs` command will generate the files libc.sa, libcurses.sa, libgcc.sa, libtermcap.sa, libdbm.sa and libgmon.sa.

The `mkimage` command actually performs the linking to generate libc.so.4.3. You will notice that the `-a, -d, -j` and `-g` switches specify the same values as in the `mkstubs` command - it is *required* that these be the same. After the `--` you basically list whatever it takes for the linker to resolve all of the symbols. In this case libc does not depend upon any external shared libraries, so we are merely linking to the libraries that were generated by the build in the libc distribution.

With `mkstubs`, you make the stub libraries as before, however you have the ability to specify additional undefined symbols that the stub library will add to each stub .o file. You used to have to specify these manually, but this is now handled more or less automatically.
An example is in order here (taken from the example directory in this distribution). The `mkimage` command used with the editlib (and the output) looks like.

First the command line

```
% mkimage -l libedit -v 0.2 -a 0x80000000 -j 0x4000 -g 0x2000 -- editline.o \
        complete.o sysunix.o -ltermcap -lc
```

Next `mkimage` identifies the version number, and it tells you the directory from which it is reading its configuration files (via the JUMP_DIR symbol)

```
mkimage v2.10
Reading configuration files from jump/
```

`mkimage` now shows you the exact command line that it will pass to the linker

```
executing:ld -x -T 80000000 -o libedit.so.0.2 __jump.o editline.o complete.o \
        sysunix.o -ltermcap -lc
```

Next comes some information from the linker. You can use these numbers to keep track of what range of virtual address your library takes up.

```
text_start: 0x80000000   text_size: 0x0000a000
data_start: 0x8000a000   gap: 0x00000000
```

Now `mkimage` shows you the sharable libraries that were linked into the new library

```
This library requires that the following shared libraries also be loaded:
8000a1ac D __NEEDS_SHRLIB_libc_4
```

Now we call `mkstubs` to generate the stub libraries. Note that we are using the same parameters for a lot of the numbers here.

```
% mkstubs -l libedit -v 0.2 -a 0x80000000 -j 0x4000  -g 0x2000 -- libedit
mkstubs v2.10
Reading configuration files from jump/
Automatically adding __NEEDS_SHRLIB_libc_4 as an undefined symbol to the
 stub libraries.
libedit.sa: ___libedit_0_200, editline, complete, sysunix, version,
        sequencing, done
```

If everything was set up properly, then you should get the stub libraries and the sharable image in your current directory.

There are other examples in the form of scripts that also appear in the doc directory of this distribution. Many lines have been commented out in various places - the people who use these simply uncomment that which they need, and leave the rest commented out.

Once you have built the library, you can install it with the following simple steps. First of all, you can copy the image itself to /lib. If you specified /lib/libtest.so as the library name, and 0.1 as the version, there should be a file libtest.so.0.1 in your current directory, and this should be copied to /lib. Next you will need a symbolic link /lib/libtest.so.0 that points to /lib/libtest.so.0.1, and this is because libraries are generally only looked up by the major number only. This makes it easy when a new minor version of the library is released because you just install the new image file and change the symbolic link, and all binaries that were linked to the old version of the library will now load the new version when they run. Finally, you should copy the files lib*.sa to /usr/lib so that you can link your programs to them by specifying the `-l` switch to `gcc`.

There is one further option to `mkimage`, `-G`, which causes an extra bit of code to be inserted for each function that has an entry in the jumptable. This extra bit of code will print the name of the function being called to stderr. You can use this facility for debugging if you are having trouble figuring out where a particular program is dying.

7. Verification

It is always a good idea to verify the stub libraries and the image after you have built them. This is a good way to catch all kinds of stupid errors of so many different kinds that is is almost impossible to enumerate them all.

To verify, use the `verify-shlib` utility supplied in the tools directory. It is invoked with a command like:

```
% verify-shlib -l libc.so.4.3 libc.sa libtermcap.sa libcurses.sa libgdbm.sa
```

You must specify an image file with the `-l` argument for `verify-shlib` to work, and you must specify *all* of the stub libraries that were generated as part of the sharable library. The `verify-shlib` program has one main task - to make sure that the absolute addresses for each symbol in the stub libraries are identical to the absolute addresses that were assigned by the linker when linking the shared image. The `verify-shlib` utility basically calls `nm` to obtain the symbol table, and then does minor editing to remove symbols that `verify-shlib` would expect to only be in the stubs or the sharable image. It then calls `diff` to obtain the differences between these lists and if everything worked correctly there should be no differences. Verify determines whether or not the diffs are null or not, and gives you a message telling you what happened. If there are differences, the `diff` output is left in a file for you to examine.
IT IS ESSENTIAL THAT ALL DIFFERENCES BE RECONCILED BEFORE YOU TRY AND USE THE LIBRARY.

In principle there would be two different types of differences. One would be where a symbol only appears in either the stubs or the image, and this actually does come up under some circumstances and is considered normal. An example of this is in libc, where there are indirect symbols in the stub libraries which tell the linker to resolve _read with a call to ___read. The `verify-shlib` program is fairly simpleminded and is not able to account for these types of symbols so they show up as differences.

Another case would be where the same symbol shows up with different addresses in the stub libraries and the sharable image.
THIS INDICATES A BUG AND MUST BE FIXED BEFORE THE LIBRARIES WILL WORK RELIABLY.
In general, `mkimage` and `mkstubs` should write the same addresses, but it is possible that a bug exists which allows there to be differences. It could also be some strange construct that was found in jump.vars or jump.funcs.

If you get a list where virtually every symbol is different, then this almost certainly indicates that you have used different addresses parameters for `mkimage` and `mkstubs`. It is ESSENTIAL that the following parameters be *identical* for all calls to `mkimage` and `mkstubs` that are used to generate components for a particular sharable library:

a) Starting text address
b) Jump table size
c) GOT size.
d) Starting data address (if used). If used for one it must be used for all.

Finally, the `verify-shlib` utility reads the image header from the sharable image to determine the address range that it will be mapped to, and this is printed to the screen.
IT IS ALSO ESSENTIAL THAT NO TWO LIBRARIES OVERLAP IN ANY WAY OR YOUR PROGRAMS WILL PROBABLY CRASH.

8. Troubleshooting

There are a lot of things that can go wrong, and if something strange happens in the `mkimage` or `mkstubs` command, the first thing you should do is check to make sure that:

1) The environment variable JUMP_DIR was properly set when compiling all of the modules of the library. The `mkimage` and `mkstubs` programs print out the path they are using - make sure this is correct.

2) The switch `-B/usr/bin/jump` was given to `gcc` for every source file that was compiled to go into the library. If this was not done, then `mkimage` or `getsize` will complain about "no source file for symbol %s" for a particular variable. (See #5 as well). Make sure that there is no trailing slash in the switch.

3) Make sure you do not have multiple copies of the same line in jump.vars or jump.funcs. A symptom of this is that when you run `mkimage`, `as` will complain about multiple definitions of the same variable in __jump.s.

4) Check to see if some program was compiled and run during the library compilation procedure. If so, all variables and functions for this program should be moved to jump.ignore. A symptom of this problem is unresolved symbols when `mkimage` is trying to link the sharable library.

5) If you are not using the LONG_FILENAME option, then you must make sure that you rebuild the library after you modify (or create) jump.vars. A symptom of this is that `mkimage` will complain about "no source file for symbol %s". This essentially means that there is no _GVAR_nnnnn.s or _GCMN_mmmmm.s file for that variable.

If a program linked to your sharable library crashes, please check the following items:

1) Make sure you specified the same parameters (virtual address, jumptable size, and GOT size) for both `mkimage` and `mkstubs`. Typical symptoms: segmentation fault when running a program - often before you reach the main() entry point.

2) Check to see if one (or more) of your libraries overlaps in virtual memory with some other library. Make 100% sure that this is not the case. The linker prints out memory usage after building the sharable image - examine these numbers and see. Typical symptoms: segmentation fault when running a program - often before you reach the main() entry point, in the __dynamic_resolve function, although it could happen anywhere.

3) Addresses in stub library do not match those in the image. Check with `verify-shlib` utility (in tools directory). Any discrepancy must be resolved before the library can be expected to work. Typical symptoms: Segmentation fault

4) Make sure that all of the shared libraries that your program needs are automatically loaded (i.e. check the `ldd` output, and see if it makes sense). In general there should never be a discrepancy here, but if there is, it may indicate a silly bug somewhere in one of the tools. If some library is not being loaded that should be, you can use the `-u` switch to `mkstubs` to force the linker to load the appropriate modules at link time. Hint: If you have to specify a library to `mkimage` in order to link the library, then it should show up with `ldd` if you selected a sharable library. Typical symptoms: Segmentation fault

*********** Message from `gcc` about not being able to find ___main**

This symbol is defined in libgcc.a, which generally comes with gcc. The reason that this entry point is needed is that it calls the constructors for all of the global objects (for modules written in C++). With older versions of gcc, this library was in /usr/lib, but with newer versions of gcc it appears in the same directory as the compiler sources. Usually at the end of the mkimage command you should have something like:

mkimage .... -lc ‘gcc --print-libgcc-file-name‘ -lc

The gcc --print-libgcc-file-name will actually print out the correct path for libgcc.a, so this will work as long as you have correctly installed gcc.

*********** Message from `gcc` about not using /usr/dll/jump.**

There are several possibilities. Check the `gcc` command line, and if `-B/usr/dll/jump/` was listed twice, this could explain the message, and the message can be ignored. If this is not the case, then you should make sure that you did not include a trailing ’/’ in the `-B` switch. Finally, you should make sure that the file /usr/bin/jumpas exists and it is the binary generated by compiling jumpas.c.

*********** Message about symbol having the wrong type, or being in the wrong file.**

You should only have variables in jump.vars, and there should only be functions in jump.funcs. These are identified by the symbol type code, "DCKT". If you have these mixed up, the tools will reject them. You have to decide whether the symbol is misidentified, or whether the symbol is in the wrong file. Here is what the various codes mean:

T - function.
C - common variable (i.e. uninitialized, or bss).
D - initialized variable.
K - constant variable (initialized data in the text section).
t - Same as T, except that entry will not be added to stub library.
U - Undefined - used as a placeholder when generating the lite C library.
A jump to a warning function is inserted in the jump slot of the jump table.

********** **Message from `as` about operands not matching a known instruction**

This indicates a bug in `jumpas`. Generate the assembly code, (i.e. make foo.s), and then do:

jumpas -k foo.s -o foo.o

The `-k` switch will leave the modified assembly code in the file jumpas.tmp, and you can try and figure out from there what the problem is and fix it.

********** **Undefined symbols from linker while running `mkimage`:**

1) First check and see if the symbols themselves are in one of the libraries. If they are, you probably need to run `ranlib`. Note that there is a bug in `ar` where sometimes `ranlib` does not generate a correct table, and this can be worked around with the commands:

```
ar -dv libc.a __.SYMDEF
ranlib libc.a
```

2) If the undefined symbol is not in any of the libraries, check to see what source file it came from (i.e. look in jump.vars and jump.funcs). Check to see if that file is in the library. Check to see if that source file was compiled properly (i.e. not a compiler error).

3) If the symbol has a __LOCAL__ at the end of it, see if a version without a __LOCAL__ is defined in one of the libraries. If it is, then `jumpas` did not run correctly. Check to make sure that you used `-B/usr/bin/jump`, and make sure that JUMP_DIR was correctly set. Make sure that this symbol is really supposed to be exported from the library in the first place (i.e. was there some program executed when building the library to generate a header file of some kind or another??)

********** **Error message: no source file for symbol '%s'**

This indicates that you have a global variable specified in jump.vars, but there is no _GVAR_nnnnn.s or _GCMD_mmmmm.s file anywhere to be found. First, find out which source module it was defined in and then see if this source file is in the library. If not, then figure out why. If this source file is in the library, then make sure that the JUMP_DIR environment variable is set properly and make sure that the switch `-B/usr/bin/jump` was specified when this module was compiled.

********** **Error message: symbol '%s' should not appear in '%s'**

This indicates that a function with a name like __GLOBAL_$[I,D]$* appears in one of the jump.* files. This should never occur, since these entry points are generated internally by `g++`, and the tools generally will filter these out. If you inadvertently add a symbol of this form to one of the jump.* files you will get this message, and you should remove this line from the jump.* file and proceed.

*********** `nm` **crashes.**

The no-cplus switch should be specified because some of the names that are used internally can confuse `nm` and make it think that the symbol is a C++ symbol which needs to be demangled.

*********** **Application linked to sharable library crashes when run.**

First of all, try and use gdb to establish that the problem is not in the application. If the problem is in the library, there are typically two ways that it will fail. It might fail at startup before the main entry point is even reached, and it might fail in some library routine for some reason. Note that just because it crashes in a library routine this does not mean that there is automatically a bug in the library - it could still be a bug in the application.

To try and isolate a bug in a sharable library, start with the following:

Make sure that the same addresses were specified to mkimage and mkstubs - use the verify-shlib utility for this purpose. If verify-shlib indicates some discrepancy, this means one of several things. There might be an error in one of the data files jump.as or jump.vars, or there could be a bug in mkimage or mkstubs.

If this indicates that everything is normal, then you will have to try and debug the library itself. If you have not stripped the sharable library, you can get gdb to load the debugging information from the file with a command like 'symbol-file /lib/foo.so.m.n'.

Keep in mind that jumpas rewrites some of the machine instructions to add indirections for any global variables that are either exported or imported. It is possible that there is a bug in jumpas that could lead to a program bug. If you suspect a bug in a particular function, then look in the file /tmp/jumpas.log. This contains a complete log of all substitutions that jumpas has made, and you can see if any of the substitutions are incorrect.

To debug a running program in assembly code, you can use the following commands to gdb:

```
1) display/i $pc     # display next machine instruction to be executed.
2) si                # step-instruction
3) info reg          # shows all machine registers.
4) disassemble       # shows disassembly of entire contents of
                     # current function.
```

If the application crashes before it even gets to main(), it indicates a problem that could be in any one of a number of places. To debug this you have to understand what happens at start time before you have any hope of fixing it. You will be able to set breakpoints in __load.c and step through to figure out exactly what is going wrong.

The first thing that happens is that all of the sharable libraries are loaded. This usually should happen without any problems coming up. Secondly, fixups are applied. These are essentially changes to the pointers in the jump table and the global offset table to reflect your usage (i.e. this is the dynamic linking). If the pointers that are handed to the program are incorrect then this will fail with a segmentation fault of some kind or another. This will probably indicate a bug in the linker of some kind or another.

Finally, builtin fixups are done. These fixups are to places in the image where a pointer variable is initialized to the address of a global variable that is being exported. The fixup tables are generated by jumpas, but a list of all of the tables is assembled by the linker. A problem here indicates either a bug in the linker or in jumpas.

Sometimes the fixup lists become corrupt, and for this reason a magic number is inserted in the header. The dynamic linking routines will check this number as they go about their business and you will get an error message if they are passed data that does not correspond to anything that they should be looking at. If you get this message, it indicates a bug that needs to be fixed.

A1. Appendix - How The Jumpas Program Works

The most critical tool that is used to generate the sharable libraries is the `jumpas` program. This effectively looks like the as assembler in terms of the inputs that it takes, and it does some minor editing of the assembler source code, and then writes it to a temporary file. The real system assembler is called to generate the final .o file, and then the temporary files are cleaned up. The `jumpas` program must be run on all of the assembly code that is destined for the sharable library, and we do this by placing the executable in /usr/bin/jumpas, and then calling `gcc` with the `-B/usr/bin/jump` switch. This makes `gcc` use /usr/bin/jump as a path prefix for various binaries, and it will thus find the assembler as /usr/bin/jumpas. The `jumpas` program takes over from here, and actually makes sure that the file is assembled after it has done its work.

There are several things that the `jumpas` program does to the input files, and these are controlled by the input files jump.vars, jump.funcs, jumpas.import and jump.ignore. The substitutions that `jumpas` will make are:

1) For each global variable declaration where the global variable appears in jump.vars, the initializer will be diverted from the assembly file and written into a separate file in the directory pointed to by the JUMP_DIR environment variable (which is also where the jump.vars file is located). Thus when `jumpas` gets through, the source file that is passed to the assembler will simply contain an external reference to the global variables that have been diverted. Here is an example:

```
.data
.globl _foo
_foo:
.long 4
```

would be the assembly language declaration for the variable foo, equivalent to the C declaration:

```
int foo = 4;
```

The `jumpas` filter converts this to something like:

```
.data
.comm _foo,0
```

which is equivalent to the C-style declaration:

```
extern int foo;
```

The `jumpas` program writes out a file _GVAR_nnnnn.s in the JUMP_DIR directory which contains the text:

```
.globl _foo
_foo:
.long 4
```

Now what have we accomplished by doing this?? First of all we have made foo an undefined symbol with respect to the library - it will not be defined anywhere. The `mkimage` program will take all of the _GVAR_nnnnn.s files and string them together with appropriate .org directives and by doing this we can always precisely control the address at which each variable will be placed. This is very important if we are to have binary compatibility from one version of the sharable library to the next.

There will also be some files _GCMN_mmmmm.s. These are the declarations for variables that are declared as such:

```
int foo;
```

without an initializer. It is important that we keep these separate from the _GVAR_nnnnn.s files because it is possible (and bad programming practice) for variables to be declared both ways in different source modules. In such cases the initialized version takes precedence.

There are two fundementally different modes of operation for `jumpas` with respect to the file naming conventions. If you are not using the minix filesystem, then you can define LONG_FILENAMES in the Makefile before you compile the tools, and this will mean that nnnnn will be the library name and the variable name. If you are using the minix filesystem, then you cannot define this symbol and the number nnnnn in _GVAR_nnnnn.s is simply the line number on which the variable in question appears in the file jump.vars. If you are using the minix filesystem, this is the main reason why it is important that you be very careful when you edit jump.vars - the data files would get scrambled up and `mkimage` would try and pull in one file thinking that it was the initalizer for _foo, and instead it pulls in _bar (`mkimage` actually checks for this and complains if there is a mismatch).

2) All functions that appear in jump.funcs will have their entry points renamed to something like foo__LOCAL__. All other references to these functions will be left alone. By making this change, we ensure that all usages of the function will go through the jump table, and thus if the startup code decides that the user is replacing a particular function we simply change the jump instruction to point to the user's function and all usage of the function will go to the users function. The library function is only referred to through the original jump instruction in the jump table, and thus we can guarantee that the library function will never get called if the user supplies a replacement function.

3) (DLL version only) All references to global variables that appear in jump.vars or jump.import will be rewritten to add an indirection to the lookup. The rewritten code is written to look at the variable __GOT__foo to find the actual address of the variable foo. The actual table is constructed when the sharable image is being created - this is an external symbol as far as the assembler is concerned when we get through with it. Let us say that we had the following machine instruction:

```
movl %eax,_errno
```

this gets rewritten as:

```
movl %ebx,__REG_SAVE_
movl __GOT__errno,%ebx
movl %eax,(%ebx)
movl __REG_SAVE,%ebx
```

The special variable __GOT__errno is assumed to contain a pointer to the variable errno that we wish to use. If the user links a program to the library that supplies a variable errno, all we need to do is modify this pointer and all references to _errno will use the storage location in the user's program. (For the time being, an entry is created in the file /tmp/jumpas.log, which shows all substitutions that it has made).

4) (DLL version only) All .long pseudo-ops that have a global variable listed as the argument, and where the global variable appears in jump.vars or jump.import, will have some extra stabs directives added to the stream which are used by the linker to generate what are called builtin fixups. Let us say that we had the following:

```
.globl _foo
_foo:
.long _errno
```

we cannot fix this through the indirect variable __GOT__errno. Instead we add an entry to a special table that tells the startup code to copy the address stored in __GOT__errno at the location _foo.

5) All global variables/functions that do not appear in any of jump.funcs, jump.vars or jump.ignore will have an entry added to jump.log. Nothing will be done to any references to these variables, but the log file is created as an aide to help you find functions that perhaps have appeared in a new version of the library that you might wish to add to jump.funcs or jump.vars.

A2. Appendix - How The Mkimage Program Works

The `mkimage` program in many ways is just a front end to the linker. The idea is that it does a little bit of preparation and then it calls the linker to do the real work.

The first job of `mkimage` is to create the jump table itself. This is just written as the assembly language file __jump.s, which contains entries like:

```
.globl _foo
_foo:
jmp _foo__LOCAL__
.align 3
```

There will be one of these jump instructions for every function listed in jump.funcs. The advantage of having a jump table is that it is easy to control the placement of an individual function's entry point and thus it is trivial to make sure that the same function has the same absolute address entry point from one version to the next. The jump table is padded with extra space at the end, and this is done to allow for future expansion - you can add more functions to the end of the jump table without moving anything else in memory around.

After the jump table comes the GOT or Global Offset Table. This simply consists of a series of pointers that look like:

```
.globl __GOT__bar
__GOT__bar:
.long _bar
```

In the section on `jumpas`, you will see that it is assumed that there will be variables of the form __GOT__bar which contain pointers to _bar. This is where they are defined. Once again, we need to specify a size for the GOT since we want to make sure that there is room for future expansion. The entries in the GOT are written in the order that the variables appear in jump.vars.

Finally, after the GOT, come the actual global variables. The `jumpas` program has diverted assembly code into files __GVAR_nnnnn.s, and now `mkimage` sucks these in one by one, looking at the variable size specified in jump.vars and adding .org (i.e. set origin) directives to make sure that exactly this much space is allowed for each variable. The global variables are defined in the order that they appear in jump.vars.

Everything up to this point has been arranged in such a way that we can guarantee that the address of each object will remain fixed from one version to the next. After the global variables come a few minor things - first of all there is a builtin fixup table. This is a list of the places that the address of a variable appears in the global data (i.e. .long _bar), accompanied by the address of the corresponding __GOT__ variable. This is used at runtime to make sure that the pointers in the global data all point to the correct locations. Finally a special symbol is inserted which will contain the address of the fixup table. This will be used at run time to make sure that all of the initialization is performed correctly.

Once this file is complete, `mkimage` assembles it, and then it basically calls the linker. The linker takes over and finishes construction of the sharable library. You can examine the file __jump.s if you wish to see what this looks like.

A3. Appendix - How The Getsize Program Works

The `getsize` program is a close cousin to `mkimage`. Its job is to figure out how large each variable is so that you know how much to allocate for it. In general, you can use the allocations that `getsize` provides, but it is a good idea to increase the size of the allocation for some structures and arrays to allow room for future expansion. The `getsize` program has no way of knowing how much padding would be appropriate, so it simply reports the facts, and it is your job to figure out how much space you actually want to allocate.

The `getsize` program works by collecting all of the data from the _GVAR_nnnnn.s files and appending them together in the order that they appear in jump.vars. It then calls the system assembler and uses `nm` to obtain a listing of the starting address of each variable. `getsize` then reads in the output from `nm`, and uses the starting address of each variable to calculate the actual size of each variable.

A4. Appendix - How The Mkstubs Program Works

The `mkstubs` program essentially generates a "library" that can be linked with a user program so that the user program will use the sharable library. It is called a stub library because each symbol does not have a traditional definition in the usual sense, but it has a special definition which essentially tells the linker that a particular symbol is at a particular absolute address. Part of the stub library is a definition of each symbol that is being exported from the sharable library and the corresponding virtual address.

There is another function that each stub library has to perform. When your program first starts up, none of the sharable libraries are mapped to your virtual address space. Before your program reaches the main() entry point, a special series of functions are called, and one of them is passed a list that the linker has generated which essentially describes which sharable libraries should be loaded. The startup code then uses the system call uselib(), which tells the kernel to add a particular library to your virtual address space. It is the job of the stub library to define a special symbol that is called a set vector (essentially the linker simply collects all of the set vectors by the same name and puts them in a list) - and this set vector is the vector that is passed to the startup code telling it which libraries need to be loaded.

Finally, the stub library must define some special symbols - it must supply definitions for the __GOT__ symbols, plus it adds symbols like __PLT__bar. These are provided so that the linker realizes that the user is redefining a particular variable, and so that the linker can make a note of the memory locations that must be updated with new information before the program can be run. The __PLT__ symbols are added so that the linker can identify a particular address as being the address of a jump instruction in a jump table - when we modify the address in a jump table it is done a little differently from when we modify the address in the __GOT__. In one case we are modifying a machine instruction, and in the other we are simply modifying a pointer.

A5. Appendix - Explanation Of How And Why Dynamic Linking Is Implemented

The first question that comes up is "Why dynamic linking - what does it do for me". The best way to answer this is to look at a simple example:

```
#include <stdio.h>

int errno;
FILE * foo;

main(){
        foo = fopen("/usr/bin/foo","w");
        printf("foo: %x errno: %d\n",foo, errno);
}
```

Assume this program is run by a non-root user. Normally you would expect that the open will fail because the user does not have permission to write to /usr/bin - this means that fopen() would return NULL, and errno would contain 13, which means EACCES - Permission denied. If you link this to a normal jump table library, it prints

```
foo: 0  errno: 0
```

Now consider the following nearly identical program:

```
#include <stdio.h>

extern int errno;
FILE * foo;

main(){
        foo = fopen("/usr/bin/foo","w");
        printf("foo: %x errno: %d\n",foo, errno);
}
```

The results for this program are:

```
foo: 0  errno: 13
```

Why does this happen???. First of all you must realize that in the first program errno is not declared external, and thus storage will be allocated in the data segment for the variable. Unfortunately there is no way to tell libc this, and libc has its own storage for errno. Thus libc writes the result in one location and you are looking in the other. In the second program errno is declared extern, and thus no storage will be allocated. The linker will see the need for a definition of errno, and it will find one in libc. Thus in the second example the user program is reading the same memory location that the library stored the answer in.

This is of course a simple example. There are more complicated examples involving programs that want to redefine a library function and these can lead to quite bizarre behavior which is difficult to debug.

The idea behind dynamic linking is that in the first program the linker will see that we are defining a variable that is also defined in libc. The linker will essentially modify libc so that it stores errno in the storage allocated in the user program, and once this happens then the test program will generate the correct result.

A5.1. Theory of operation:

Basically, to implement dynamic linking we need to route all function calls to global functions through the jump table. We also need to make sure that all global data is accessed indirectly through a pointer. The actual libraries do not really look very much different from the actual jump table libraries that we are currently using, but there are additional pointers to the global data in the sharable image that are used internally by the library to access all global data. There are also additional symbols in the stub library that are defined as the address of the various pointers, and are used by the linker to help determine whether there is a pointer or a jmp instruction that needs to be fixed up.

In principle we could use `-fPIC` with `gcc` to generate the additional indirections, but this has a number of nasty side effects, and I decided to forego PIC. Instead the `jumpas` program automatically converts any references in the code and adds the required indirections. The indirections are always referred to through variables like __GOT__foo, which are simply pointers to _foo. The `mkimage` tool will automatically generate the table of pointers and position it properly so that all externals will be properly resolved. Similarly the `mkstubs` tool will generate a stub library with all of the symbols properly defined to point to the analagous objects in the sharable image.

The linker does something else important, however. It watches for duplicate definitions of symbols, and if the old symbol is at an absolute address, it makes a note of it as a conflict that needs to be resolved at run time. The linker puts a list of all of these conflicts into a table and makes it available to crt0 through the variable __SHARABLE_CONFLICTS_. The linker assumes that any symbol in the sharable library that starts with a __GOT_ is a pointer to the same variable without the __GOT_. It is also assumed that a symbol with a __PLT_ in front of it indicates that there is a jmp instruction there. The linker goes through each of the symbols that are locally defined and looks for __GOT_ and __PLT_ variants of the same symbol name, and any of these would indicate that there is a pointer or a jump instruction in the sharable library that needs to be fixed at run time.

At run time, crt0 (through the function __dynamic_resolve) will go through and fix up all of the pointers that need to be fixed. The fixup essentially just consists of copying one long integer from one address to another, so the actual runtime overhead should be negligible. Note that when a sharable image is linked, a list of fixups will be made in that image as well, and those fixups need to be run as well as the ones in the main user program.

A6. Appendix - Example of building compatable DLL libraries

Here is an example of building a compatable DLL library if you have no jump.* files. Let's choose to build a Xaw compatible library with Xaw3d source. Firstly you must have the stub of the library you are trying to be compatible with. Obviously libXaw.sa.

```
% libinfo /usr/X386/lib/libXaw.sa
```

You should get something like this returned

```
Library requires libX11.so.3
Library requires libXt.so.3
Library requires libc.so.4
Library requires libXaw.so.3
Library version is 3.0pl2
Start address of table is 0x60300000
Looks like this is part of libXaw.so
Currently GOT size is 0x1000 (4096)
Jump table size is 0x4000 (16384)
Encountered 84 PLT's and 73 GOT's in stub
```

Ok so now you have the `-a, -j, -g` and `-v` flags for `mkimage` and `mkstubs`.
☞ **Warning**: DLL libraries built with tools prior to version 2.4 may give incorrect values for the GOT and Jump table size. You will get a warning to this effect. Generally if the stub you are inspecting is part of a larger library merged with various smaller libraries, (e.g. libX11.sa), then you should, for example, run the libinfo command on libX11.sa instead of libXmu.sa.

Now do a first pass compile with fresh source to produce a jump.log file in the directory pointed to by JUMP_DIR. Generate the jump.vars file with the `-v` flag to mkcompat:

```
% mkcompat -v /usr/X386/lib/libXaw.sa > j.v
Found 13 new global(s)
```

And now get the jump.funcs file via the `-f` flag.

```
% mkcompat -f /usr/X386/lib/libXaw.sa > j.f
Moved 17 new function(s) to the bottom.
```

This shuffles the ordering of the globals to be compatable with the old stub, and adds the reserved sizes found in the GOT for all the jump.vars it knows about. If in the new source you lost some vars/functions, automatic U or __DUMMY__ definitions will be placed in the address space and a message like

```
Used 2 placeholder(s) for lost function(s)
```

will be output to stderr.

Theoretically you can redirect the output straight to jump.funcs/vars. However always check to see if any lines were taken out or inserted as a result of a bug in the program. When you are satisfied move the file j.f to jump.funcs and j.v to jump.vars. It is also possible to extract a jump.import file by looking at the "Library requires foo.so" lines produced by libinfo.

There is a problem when attempting to get the reserved size of the last global as it has the rest of the GOT to play with. In theory, even when building DLL libraries with the standard tools, you are justified in using the *exact* value determined by `getsize` for the *last* global. You only have to think hard about appropriate padding when new symbols get added and the last global becomes an intermideate one. For this reason the file generated by the command "*mkcompat -v*" will have the lines

```
# The following symbol was found in the old stub but its size
# could not be determined as it is the last symbol in the table.
# Please use getsize to determine its size.
00000000 D _viewportWidgetClass libXaw        Viewport
```

You may also have the following lines if new globals are found.

```
# Anything below this line is new to the old library
# Note: It is possible that some/all of these could be moved to jump.ignore
00000000 D _layoutClassRec       libXaw        Layout
00000000 D _layoutWidgetClass   libXaw        Layout
00000000 D _smeThreeDClassRec   libXaw        SmeThreeD
00000000 D _smeThreeDObjectClass libXaw       SmeThreeD
00000000 D _threeDClassRec       libXaw        ThreeD
....[more deleted].....
```

This is where you decide whether to create a jump.ignore file or leave them in jump.vars as new symbols. A similar procedure must be carried out for jump.funcs if new functions are detected. If you decide all new globals should remain then you need do nothing else as the `mkstubs` and `mkimage` programs accept comments in the jump.vars and jump.funcs files.

Now re-compile for the second pass. Use `getsize` as usual to determine the size of the last and/or new globals. Then do the standard `mkimage`/`mkstubs` commands with the information you obtained from running libinfo above. As is always good practise, use `verify-shlib` to verify the new library. Et voila, drop in your new shared library if all is OK.

A7. Appendix - Short Summary Of Programs

Program: jumpas

Input files:
jump.vars Global data to be exported.
jump.func Global functions to be exported.
jump.ignore Globals not to be exported
jump.import Globals imported from another library.

Output file:
jump.log All globals not listed in one of the above 3 files.
_GVAR_nnnnn.s Declaration for initialized variable.
_GCMN_nnnnn.s Declaration for common variable.
/tmp/jumpas.log Listing of all of the assembly code rewrites that are performed in order to add indirections to the global variables being imported or exported.

The input and output files are in the current working directory unless the environment variable JUMP_DIR is set. If JUMP_DIR is set, it is assumed to be a prefix to be prepended in front of all of the filenames. For example, if you set it to /tmp/ then all of the temporary files will go into /tmp. The form of the GVAR and GCMN names depends upon whether the LONG_FILENAMES option is specified.

Program: mkimage, mkstubs

Command line options:
-j nn Memory to reserve for jump table in bytes.
-g nn Memory to reserve for GOT in bytes.
-v version Version number to store in jump table. This should be in the format M.N or M.N.P
-a offset Memory address for sharable library
-d offset Memory address for data section in sharable library (only used with libc).
-l outlib Specification of output library name. It should look something like "/lib/libfoo". shr-build will append the .so.<major> automatically.
-u symbol (mkstubs only). Add an external reference to symbol in each module in the stub library.
-G (mkimage only) Generate a version of the library that will print the name of each function on stderr as it is being entered. Useful for debugging.
-f Force changes in parameters.

-- Indicates the end of the options for shr-build. Anything after this will be passed to the linker when creating the sharable image.
<linkopt> You must specify whatever libraries and object files are needed in order to link the sharable image.

Input-files (in cwd or JUMP_DIR):

jump.vars	Global data to be exported.
jump.funcs	Global functions to be exported.
jump.undef	(mkstubs) List of other libraries which need to be loaded with this one is loaded.
_GVAR_nnnnn.s	Declaration for initialized variable number n.
_GCMN_nnnnn.s	Declaration for common variable number n.

Output files:

__jump.s	Jump table and global data for jump library.
jump.undef	(mkimage)

Example:
```
mkimage -l /lib/libtest -a 70000000 -v 3.2 -j 4096 \
    -- myfile.o mylib.a -lc ‘gcc --print-libgcc-file-name‘ -lc
```

This will build the stub library libtest.a and the sharable jump-table library libtest.so.3, both of which will be left in the current working directory.

A8. Appendix - General Rules for editing jump.*:

1) If you are not using the LONG_FILENAMES option to the tools, you have to be very careful about how you change the file jump.vars. *Any* change other than modifying the sizes in jump.vars will mean that you will have to recompile the library. An exception is if you are adding a second library to the original, and you add symbols to the *end* of jump.vars and jump.funcs.
2) If you change the order of the entries at all (with the exception of adding new entries at the end), you should delete all of the _G*.s files and then rebuild the library.
3) If you only change an offset for a variable, (without changing the order of any lines, deleting any lines or adding any lines), you can just run `mkimage` or `mkstubs` again. You do not need to recompile the library.
4) Two different data items cannot overlap in memory. This means that you must specify a size that is the same size or larger than the actual size of the variable. The assembler will crash if you attempt to do this.

If you are using the LONG_FILENAMES option, then you have a lot more flexibility. The only time you will need to recompile the library is if the library source code changes.

Table of Contents