Apache Shared Modules

by Brian Long

One of the potentially exciting things about Kylix is being able to write Apache web server shared modules. However, it can be frustrating, so my goal in this article is to explain what you need to do to write shared modules and to get them to work with Apache. I should warn you that it may be more complicated and involved than you might expect...

First things first. Kylix can generate applications for Apache version 1.3.9 and later, running under Linux. Also, the support for generating Apache applications is restricted to the Server Developer edition and the Enterprise Studio edition (it's not in the Desktop Developer or Open Edition).

I should also mention that the name *Apache* Web Server comes from the fact that it started life as *a patchy* web server, based on the code base for another popular web server.

One final thing: this article will not discuss how to write web applications, since the Windows Delphi WebBroker concepts all transfer readily across to Kylix and Linux. For more information on how WebBroker applications work, see past and future editions of Bob Swart's *Under Construction* column.

Apache And CGI Applications

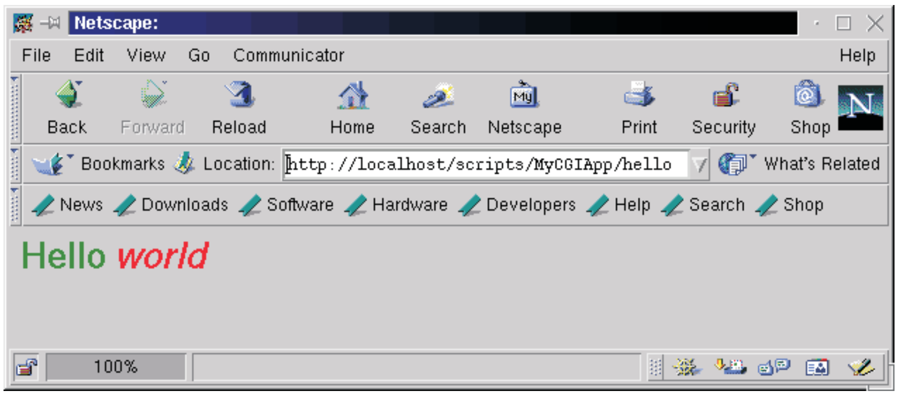
Kylix supports making CGI applications which Apache can make use of, in just the same way that Delphi can generate CGI applications for Microsoft PWS (Personal Web Server) or IIS (Internet Information Server). For completeness, I'll just briefly run through what is needed for CGI applications to operate correctly.

Getting a Kylix CGI application to work is quite straightforward, since the Kylix README file explains how to modify the Apache configuration file to support running CGI applications. It's a case of making sure it contains a `ScriptAlias` directive that maps a logical script directory to a physical directory that will contain CGI applications. Additionally, you must add in a `Directory` directive that specifies the `ExecCGI` attribute. Listing 1 shows both these directives.

➤ *Listing 1: An extract from httpd.conf (the Apache configuration file).*

```
ScriptAlias /scripts/ "/<path>/ cgi-bin/"
<Directory "/<path>/cgi-bin">
  AllowOverride None
  Options ExecCGI
  Order allow,deny
  Allow from all
</Directory>
```

➤ *Figure 1: A simple CGI application running from Apache with output in Netscape.*


Netscape:
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop
Bookmarks Location: http://localhost/scripts/MyCGIApp/hello What's Related
News Downloads Software Hardware Developers Help Search Shop
100%

Hello world

Since your CGI applications will need to be placed in this directory, either ensure you have rights to create files there (use `chmod`, as explained later), or be prepared to copy your CGI binaries into the directory whilst logged in as `root`.

You must also make sure that the library files required by your CGI program will be found. This is usually done by adding a `SetEnv` directive to the Apache config file to set the `LD_LIBRARY_PATH` environment variable, but can also be achieved in other ways (see the *Ensuring Libraries Can Be Found* boxout). On a development machine, this variable should point to the Kylix `bin` directory, but when deployed, the directive may not be required depending where the libraries are placed, as explained in the boxout.

Note that the Kylix README file also tells you to use the `SetEnv` directive to set the `LANG` environment variable, but in fact this step is unnecessary with the shipping product. During Kylix development, setting this variable ensured known problems with certain releases of the Linux `glibc` library could be avoided. The shipping product overcomes these problems by calling the `CheckLocale` routine in the `SysUtils` unit initialisation code, which ensures that the locale details are set correctly to avoid these problems, regardless of whether the `LANG` variable has been set or not.

Figure 1 shows a trivial CGI application (MyCGIApp) responding to a `PathInfo` value of `/hello`. If you wish to use dbExpress in a CGI application, check the *dbExpress Configuration Information* boxout.

Apache And Shared Modules

Kylix also supports making Apache shared modules, which are similar to ISAPI DLLs for Windows web servers. Shared module functionality lives in shared object files (.so files, which are the Linux equivalent of Windows .dll files). This is what we will concentrate on for the rest of the article.

A Windows DLL can be statically linked to an application or it can be dynamically loaded at runtime. If it is statically linked, the linker inserts references to the DLL in the application's file header. Dynamically loaded DLLs are not known by the linker, and are loaded under program control with the `LoadLibrary` Windows API. IIS and PWS can dynamically load an ISAPI DLL when the name of the DLL is specified in a URL.

Similarly, code in a Linux SO file can be statically linked or dynamically loaded. Much functionality is made available in default modules (SO files) which are statically linked, and these are known as *shared libraries* or *DSO libraries*.

Kylix supports generating Apache functionality in SO files which are dynamically loaded by Apache with the `dlopen` Linux API through references added to its configuration file. These libraries are called *shared modules* or *DSO files* (as opposed to DSO libraries).

The Problem

Unfortunately, it seems Apache does not have shared module support enabled by default. Instead, the available modules that are installed on your behalf are all statically linked to Apache. To enable this support we must recompile Apache with specific options. This will apply both on your development machine and also on any machines you deploy to.

Incidentally, if the thought of recompiling your web server makes you nervous, then rest assured that the process is reasonably painless if you follow the directions.

The subject of shared modules and recompiling Apache to work with them is discussed in the Apache documentation which, if installed, is at http://localhost/manual/dso.html whilst Apache is running.

On SuSE Linux, the Apache binaries and documentation are both installed from the same package. However, Red Hat Linux has the

Ensuring Libraries Can Be Found

The primary problem in getting Kylix applications to run is ensuring the Linux dynamic library loader (ld.so) knows where to locate any library files required by the program.

When developing applications, the Kylix README explains that we can solve the problem by running the `kylixpath` script. This script sets up a number of environment variables (`PATH`, `LD_LIBRARY_PATH`, `XPPATH` and `HHHOME`), the key one for this discussion being `LD_LIBRARY_PATH`. This variable lists directories that should be searched by the library loader to locate dynamically loaded library files.

Using the `kylixpath` script works well whilst developing normal Kylix applications, but it does not apply so well to developing CGI web applications or Apache shared modules, nor does it apply to ensuring applications work when they have been deployed.

Whilst developing any Kylix application, all the key library files are in Kylix's `bin` directory (including the MIDAS library, libmidas.so.1, as used by dbExpress). If you installed Kylix as a regular user, this means the library files are located somewhere under your home directory. If you installed as the `root` user, they will be either somewhere under the `/root` directory, or in a more generally accessible file system branch, such as under `/opt` or `/usr/local`.

The problem of telling the loader where to find libraries is typically solved in one of two ways. You either place the libraries (or symlinks to them) in locations already known to the loader, or inform the loader where to look for the libraries.

Placing Libraries In Known Locations

The loader always looks in `/lib` and `/usr/lib`, so placing the libraries (or symlinks to them) in either of these directories will work. To create a symlink in `/usr/lib` use:

```
ln -s <path to library>/<library name> /usr/lib/<library name>
```

Informing The Library Loader Where To Look

If you prefer to instruct the loader where to find the libraries, then either the user that will run the application sets the `LD_LIBRARY_PATH` variable to include the library paths, or the library paths are added to the dynamic loader configuration file and cached.

You can modify any user's library search path by editing that user's ~/.bashrc file (the .bashrc hidden file in their home directory) or even ~/.bash_profile on Red Hat systems, if it exists, to contain what is shown below, specifying the library path where indicated:

```
if test -n "$LD_LIBRARY_PATH" ; then
  LD_LIBRARY_PATH="$LD_LIBRARY_PATH:<library path>"
else
  LD_LIBRARY_PATH="<library path>"
export LD_LIBRARY_PATH
```

This assumes the user runs the Bash shell, which is the default shell used on Linux.

One important point about this option is it relies on the user running the program having a home directory and a customisable environment. When Apache runs an application, it runs as user `nobody` which does not have a home directory. In order to set the `LD_LIBRARY_PATH` variable for a CGI program, you must add a `SetEnv` directive into the Apache configuration file, httpd.conf (as described in the Kylix README file):

```
SetEnv LD_LIBRARY_PATH <path to libraries>
```

Unfortunately, the `SetEnv` directive only applies to CGI applications. It will do nothing for shared modules.

The alternative to changing the library search path variable is to add the library directory (or directories) into the /etc/ld.so.conf file (on SuSE Linux you do this indirectly by adding the location to /etc/ld.so.conf.in and then running `SuSEconfig`). You then run the `ldconfig` program which reads /etc/ld.so.conf and creates a cache file (`/etc/ld.so.cache`) based on all the libraries it finds in the listed directories. All these libraries are then automatically available.

Apache documentation in a separate package to the binaries. To verify whether the documentation has been installed on Red Hat, run:

```
rpm -q apache-manual
```

The Source

The first thing to do is obtain the Apache source code. If you happen to be using Red Hat Linux 7.0, you can find it on the Source CD in the SRPMS directory. Alternatively, you can download it from

```
http://httpd.apache.org/dist/
  apache_1.3.14.tar.gz
```

During this process of modifying Apache it is important to be logged in as `root` so that you have enough rights to remove and add software when appropriate.

Next, you need to uninstall your current Apache installation (assuming you have one). To check whether Apache is installed, run:

```
rpm -q apache
```

If you are told that Apache is present, locate and backup your Apache configuration file and then uninstall Apache with the commands in Listing 2.

Now you should unpack the source into some suitable location (such as `/usr/src`) as shown in Listing 3. This makes a directory called `/usr/src/apache_1.3.14` containing the source code tree.

➤ *Listing 3: Unpacking the Apache source.*

```
cd /usr/src
tar xvzf apache_1.3.14.tar.gz
```

The Plan

The principal goal is to compile Apache with some options that affect the layout of the main binary file, httpd. Rather than this being one monolithic file, we need to get most of its code placed in a shared module called libhttpd.so and an accompanying executable program libhttp.ep. The httpd file will end up containing nothing much more than bootstrapping code.

Additionally, Apache must be compiled with the `so` module enabled (this will be statically linked into Apache). You can enable as many other standard modules as you like, but it is vital that `so` is enabled.

Finally, Apache must be linked with the `pthread` library to overcome a Linux loader bug where Apache refuses to run any Kylix-generated shared objects. You can verify that Apache has not been linked with the `pthread` library by default by running the `ldd` command across the main httpd binary.

➤ *Listing 4: The custom script to build Apache with DSO support.*

```
#!/bin/sh
##
## Use this shell script to re-run the APACI configure script for
## restoring your configuration. Additional parameters can be supplied.
##
LIBS="/usr/lib/libpthread.so" \
./configure \
"--with-layout=Apache" \
"--enable-module=so" \
"--enable-rule=SHARED_CORE" \
"$@"
```

```
locate httpd.conf
cp <path>/httpd.conf ~/httpd.conf.bak
rpm -e apache
```

➤ *Listing 2: Making a backup of the config file before removing Apache.*

There will be an httpd script in /etc/init.d (Red Hat or Debian) or /etc/rc.d/init.d (Red Hat), but the actual httpd binary file that gets invoked by it will be located elsewhere, perhaps in /usr/local/apache/bin or /usr/sbin. Running `ldd httpd` will produce a list of all the libraries linked to httpd. Where the binary is linked to a library file which is actually a symlink, the real library file will also be shown.

Putting The Plan Into Practice

We can accomplish all these things by passing appropriate options to the `configure` script that is at the root of the source tree. However, we can also simplify this by making a config.status script (if one is not already present) that acts as a front end for `configure`. If the file is not already there, you can make it like this (the second command allows the resultant script file to be executed by everyone).

```
touch config.status
chmod a+x config.status
```

config.status should look like Listing 4. The `SHARED_CORE` rule gets the

➤ *Listing 5: Two Apache directory layouts from config.layout.*

```
#   Classical Apache path layout.
<Layout Apache>
    prefix:        /usr/local/apache
    exec_prefix:   $prefix
    bindir:        $exec_prefix/bin
    sbindir:       $exec_prefix/bin
    libexecdir:    $exec_prefix/libexec
    mandir:        $prefix/man
    sysconfdir:    $prefix/conf
    datadir:       $prefix
    iconsdir:      $datadir/icons
    htdocsdir:     $datadir/htdocs
    cgidir:        $datadir/cgi-bin
    includedir:    $prefix/include
    localstatedir: $prefix
    runtimedir:    $localstatedir/logs
    logfiledir:    $localstatedir/logs
    proxycachedir: $localstatedir/proxy
</Layout>
```

```
#   RedHat 5.x layout
<Layout RedHat>
    prefix:        /usr
    exec_prefix:   $prefix
    bindir:        $prefix/bin
    sbindir:       $prefix/sbin
    libexecdir:    $prefix/lib/apache
    mandir:        $prefix/man
    sysconfdir:    /etc/httpd/conf
    datadir:       /home/httpd
    iconsdir:      $datadir/icons
    htdocsdir:     $datadir/html
    cgidir:        $datadir/cgi-bin
    includedir:    $prefix/include/apache
    localstatedir: /var
    runtimedir:    $localstatedir/run
    logfiledir:    $localstatedir/log/httpd
    proxycachedir: $localstatedir/cache/httpd
</Layout>
```

httpd binary split up as described above. The `with-layout` option specifies which directories will be used by the various Apache files. Listing 4 specifies the default layout, where Apache is installed in /usr/local/apache and the configuration file is /usr/local/apache/conf/httpd.conf.

The typical Red Hat Apache installation is compiled with the `RedHat` layout instead, where Apache is installed in /usr, with binaries going in /usr/bin and /usr/sbin and the configuration file is /etc/httpd/conf/httpd.conf.

You can view all the available directory layouts by examining the config.layout file. Listing 5 shows the Apache and `RedHat` layouts defined. You should be able to see how the directories get built up from the individual entries. Any directories mentioned from this point on will assume the default Apache layout.

Note that the original Kylix 1.0 help also describes this compilation process (in *Apache DSO applications, compiling*), and shows a suggested config.status file. However, the version in the help erroneously includes two extra parameters compared to Listing 4.

It passes `CFLAGS="-g"` and `CFLAGS_SHLIB="-g"` which tell the `gcc` compiler to include full debugging information in the compiled binary. This will swell the Apache binary considerably and may also disable optimisations and enable assertions, and so should not be used in a production system.

To compile and install Apache from the source requires three commands:

```
./config.status
make
make install
```

The first generates the make files, the second builds Apache and the third installs it according to the requested directory layout.

Running `ldd httpd` (where `httpd` is likely to be in /usr/local/apache/bin) will now show that Apache is linked with the `pthread` library. Listing 6 shows the command being executed, after changing to the right directory, along with its output. Note that Linux allows you to enter multiple commands at the same time if you separate them with semicolons.

➤ *Listing 6: ldd verifying the pthread library has been linked.*

```
[blong@Cube blong]$ cd /usr/local/apache/bin; ldd http
   libpthread.so.0 => /lib/libpthread.so.0 (0x40020000)
   libm.so.6 => /lib/libm.so.6 (0x40036000)
   libcrypt.so.1 => /lib/libcrypt.so.1 (0x40055000)
   libdl.so.2 => /lib/libdl.so.2 (0x40083000)
   libc.so.6 => /lib/libc.so.6 (0x40087000)
   /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

➤ *Listing 7: Starting Apache with a boot script.*

```
#for RedHat:
/etc/rc.d/init.d/httpd start
#for Debian or Red Hat (Red Hat has a symlink):
/etc/init.d/httpd start
#for SuSE:
/sbin/init.d/apache start
```

➤ *Listing 8: The shared module project file.*

```
library Foo;
uses
   WebBroker, ApacheApp,
   FooU in 'FooU.pas' {WebModule1: TWebModule};
{$R *.res}
exports
   apache_module name 'Foo_module';
begin
   Application.Initialize;
   Application.CreateForm(TWebModule1, WebModule1);
   Application.Run;
end.
```

With the default Apache layout, shared modules are found in /usr/local/apache/libexec (/usr/lib/apache in the `RedHat` layout). Since your shared modules will also be placed there, you must either copy them there whilst logged in as `root` or give yourself write access to the directory, for example with:

```
chmod o+w
  /usr/local/apache/libexec
```

Having built a suitable version of Apache, you should make sure it is running. Different distros differ in how you ensure it runs by default at startup, but you can always start it manually with the apachectl script in /usr/local/apache/bin. Run this command to start Apache:

```
apachectl start
```

Alternatively, you can run the boot script (described earlier for Red Hat and Debian) with a `start` parameter, using one of the commands in Listing 7.

If Apache fails to start, look in the error log file /usr/local/apache/logs/error_log to try and find out why.

With Apache (hopefully) successfully running we can move onto the business of actually creating shared modules. There are three steps to having a successfully operating Apache shared module: first create the shared module, second load the shared module, and finally set the module handler.

Creating The Shared Module

Choose `File | New...`, then `Web Server Application`, then `Apache Shared Module (DSO)`. This makes a project much the same as a CGI application, but with a different project source file (see Listing 8). Firstly, a shared module is a library, not an application. Secondly, there is a symbol exported by the project file. This basically means that building the functionality for a shared module is much the same as for a CGI application. It's just the project file that differs.

The sample project file in Listing 9 has been saved as Foo.dpr. The exported symbol, `apache_module`, is actually the name of a record containing information that Apache needs. Kylix therefore now allows data items to be exported as well as just procedure and functions. It is typical for a shared module to rename this symbol during the export process and the expected pattern is `<project_name>_module` (although any pattern is valid). A project called `Foo` therefore typically exports the symbol `Foo_module`.

There are four key strings to understand when setting up a shared module.

1. The module record name, `Foo_module`. This exported symbol specifies how the module description record will be known to Apache. Apache finds all the information about your shared module from this record. This is case sensitive, so make sure you get the right case when referring to it.

2. The library name, libFoo.so. This is the name of the resultant binary file. Shared objects default to having a .so suffix and a lib prefix (these can be changed in the project options dialog, or with compiler directives).

3. The module name, `libFoo`. This defaults to the base name of the resultant binary library file. This can be changed by assigning a value to the `ModuleName` variable before the call to `Application.Initialize` (either in the project file, or in a unit initialisation section). This string is placed in the `name` field of the module record.

4. The content type, `libfoo-handler`. This defaults to the module name in lower case with a `-handler` suffix. This can be changed by assigning a string to the `Content-Type` variable before the call to `Application.Initialize`. This string is stored as the `content_type` field in the `handlers` record field of the module record. The Apache source specifies this value should be lower case, so you should respect this rule.

This library should be compiled to the same directory where other shared modules reside which, in the default Apache directory layout, is the libexec directory under Apache's installation directory. Remember to make sure you have rights to create files in this directory, as explained earlier.

If you get a warning from Kylix during compilation/linking saying it cannot find libhttpd.so, this is because the library cannot be found on any of the normal library search paths (it's in the shared modules directory, /usr/local/apache/libexec). To allow Kylix to see the library, follow one of the suggestions in the *Ensuring Libraries Can Be Found* boxout.

The shared module should now compile without any warnings. Our next task is to ensure that it will run successfully from Apache. We have the same problems as we encountered with CGI applications, in that certain required libraries (notably the MIDAS library, if you are using dbExpress) will not be found by the library loader. Again, you can resolve this with the steps described in the *Ensuring Libraries Can Be Found* boxout.

When your application is deployed, make sure that the global dbExpress configuration files are used, or alternatively compile all of the driver and connection parameters into the application (see the *dbExpress Configuration Information* boxout for details of how to do this).

dbExpress Configuration Information

A dbExpress application needs to connect to a supported database. dbExpress applications typically use MIDAS and so the MIDAS library, libmidas.so.1, must be deployed with the application and made available. The details about the database connection and the driver needed to make it must also be available to the dbExpress application in order for it to work correctly. This is perhaps most cleanly done by compiling them into the application, although this means the location of the database (and all the other connection/driver details) cannot change. If this is acceptable, ensure the `TSQLConnection` component's `Params` property has all the required parameters set up at design-time, and that its `LoadParamsOnConnect` property is `False`.

If you prefer to use configuration files, you must understand how dbExpress locates them. Inside the `SqlExpr` unit (which implements `TSQLConnection`) a private `GetRegistryFile` routine exists. If `LoadParamsOnConnect` is set to `True`, this routine is used to locate the configuration files. It first looks in the logged in user's home directory to see if there is a hidden .borland directory which, if a Kylix developer is logged in, there will be. Inside ~/.borland the code uses the dbxdrivers and dbxconnections files.

The idea is that individual developers can use Kylix on the same system and keep their own dbExpress settings separate from each other. During development, you can make sure your CGI application uses your own dbExpress settings by adding another `SetEnv` directive to Apache's configuration file to tell it which home directory to look in for the .borland directory (see below). However, this should only be done during development. Giving the nobody user a valid `HOME` variable is never advisable in a production system.

```
SetEnv HOME <home directory>
```

When you deploy your CGI application, you cannot assume that any particular user will have appropriate dbExpress configuration files. So, instead of user-specific dbExpress configuration files, you should use the global configuration files. Kylix installs these in /usr/local/etc as dbxdrivers.conf and dbxconnections.conf and you should deploy your settings to the same files (taking care in case the files already exist).

When the dbExpress `GetRegistryFile` routine looks to see if `$HOME/.borland/` exists and finds that it doesn't on a production system, it will fall back to using the global configuration files which should always be made available to it. However, it should be noted that using the global configuration files always makes a failed call to `FileExists` followed by a successful call to `FileExists` for every request.

You must weigh up the pros and cons of compiling the dbExpress parameters into the shared module, meaning you cannot move the database, versus using the global configuration files, which means redundant `FileExists` calls (which themselves call the Libc `euidaccess` API) being made.

Load The Shared Module

In httpd.conf, there is a (possibly empty) section set up for loading shared modules. Each shared module gets a `LoadModule` entry in the list set up like this:

```
LoadModule module_record_name
  library_name
```

If the library is not in the Apache root directory (which of course it isn't) you must include relative path details. In the default directory layout, this will mean specifying something like:

```
LoadModule Foo_module
  libexec/libFoo.so
```

Note again that the module record name is case-sensitive, so this version would *not* work:

```
LoadModule foo_module
  libexec/libFoo.so
```

Set The Module Handler

In order for your shared module to handle various requests you set up a location attribute (typically immediately below the `LoadModule` directive) and specify a handler for it in terms of a content type. Listing 9 shows the generic layout of the `Location` directive, along with an example that specifies all requests that start with the path /kylix are to be handled by your module.

This tells Apache that all requests with a /kylix path should be handled by your module (remember that the content type should be lower case). Assuming you have a default action in your shared module, you can invoke it with the URL http://localhost/kylix. Figure 2 shows this situation where the shared module is responding to a `PathInfo` of /cust.

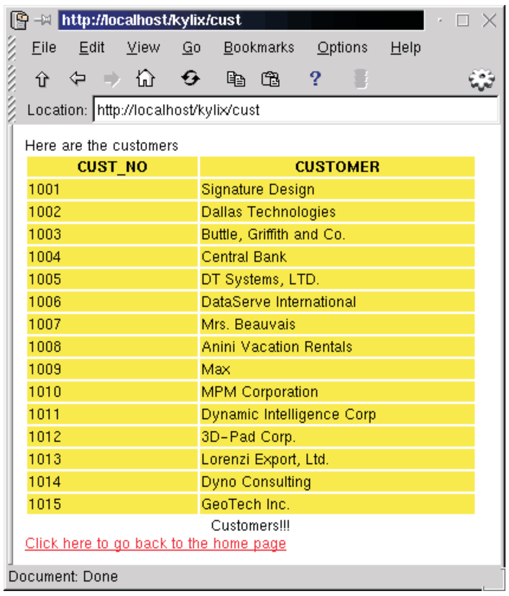
Common Errors

Whilst trying to get shared modules to work, you may come across a number of obstacles, many of which are reported in the Apache error log file. Here are some explanations of the more common problems, where each resolution has been implemented throughout the preceding text:

➢ The CGI app gives the *Internal Server Error* message in the browser. This indicates that the required `SetEnv` CGI directives are not in the Apache configuration file, or Apache has not been restarted since adding them.

➢ During compilation, Kylix gives a warning *File not found: 'libhttpd.so'*. This indicates the (required) file is not on the library search path. If the file does exist somewhere in the file system, set up a symlink in a standard library directory, such as /usr/lib. If the file does not exist, you must recompile Apache with the `SHARED_CORE` rule to create it.

➢ The error log says: *child pid <pid> exit signal Segmentation fault (11)*. This can be caused by the DSO not being able to locate the MIDAS library, libmidas.so.1. Unfortunately, the dbExpress error checking for this problem is non-existent and ultimately the code jumps to a `nil` function pointer as a result. Since Apache runs as user `nobody`, you must make sure that this library is accessible for that user. Making a symlink to it from a standard library directory, such as /usr/lib, can remedy the problem.


http://localhost/kylix/cust
File Edit View Go Bookmarks Options Help
Location: http://localhost/kylix/cust

Here are the customers

CUST_NO	CUSTOMER
1001	Signature Design
1002	Dallas Technologies
1003	Buttle, Griffith and Co.
1004	Central Bank
1005	DT Systems, LTD.
1006	DataServe International
1007	Mrs. Beauvais
1008	Anini Vacation Rentals
1009	Max
1010	MPM Corporation
1011	Dynamic Intelligence Corp
1012	3D-Pad Corp.
1013	Lorenzi Export, Ltd.
1014	Dyno Consulting
1015	GeoTech Inc.

Customers!!!

Click here to go back to the home page

Document: Done

➤ *Figure 2: A shared module displaying output in the KDE File Manager.*

```
# The generic layout of the
# Location directive
<Location /location>
  SetHandler content_type
</Location>
# An actual example of the
# Location directive
<Location /kylix>
  SetHandler libfoo-handler
</Location>
```

➤ *Listing 9: Setting the module handler.*

➢ The error log says: *loaded DSO <module name> uses plain Apache 1.3 API, this module might crash under EAPI! (please recompile it with -DEAPI)*. This is an indication that Apache is not compiled with DSO support, or that the `so` module is not enabled. Follow the instructions to rebuild Apache as above, making sure you have definitely removed any previous version of Apache that may have been installed.

➢ The error log says: *shared object not open*. This is usually because DSO support has not been enabled and the module fails to find libhttpd.so. You must recompile Apache with DSO support enabled as directed above.

Finally

Bob Swart will come back to the subject of Kylix CGI applications and shared modules for Apache in his *Under Construction* column next month. Until then, I hope the information presented here helps get you started playing with Apache shared modules under Linux.

Acknowledgements

Thanks are due to Shane Hausle from Borland QA for much needed help getting through a couple of Apache problems, and suggestions for improving the clarity and accuracy of this article.

Brian Long is a freelance trainer and problem solver specialising in Delphi, Kylix and C++Builder work. Visit www.blong.com or email him on brian@blong1.com