

# Write software for multiple UNIX platforms

Skill Level: Intermediate

[Martin Brown \(questions@mcslp.com\)](mailto:questions@mcslp.com)

Freelance writer

Consultant

21 Feb 2006

If you write software for more than one UNIX® platform, you are aware of the difficulty of getting your software to compile on two platforms. This tutorial covers tools and tricks that can make the process of supporting different UNIX platforms significantly easier at the code level. The reason is not the lack of tools or a binary compatibility issue, but rather the problems with the header files and functions that set up a standard UNIX environment.

## Section 1. Before you start

In this tutorial, you're going to look at the issues surrounding both building and migrating your applications for compatibility across a wider number of UNIX® platforms. Rather than concentrating on specific platform differences, you'll instead be given the knowledge and tools to make decisions and determinations for yourself to achieve cross-compatibility with your UNIX applications.

### About this tutorial

Developing software that compiles and builds on multiple platforms can be a complex task. There are minor, and major, differences between the different UNIX variants that can cause problems. These range from missing tools and libraries, to differences in the header files that you use to build your code. Accounting for these differences make your code more portable, which is the focus of this tutorial. You'll also look at using GNU autotools, which can take away some of the pain and complexity in the migration and development process.

## Prerequisites

You'll need access to a C compiler on your system to try out some of the examples. To try out the example autotools session, you need access to the Autoconf/autotools packages available through the [GNU Web site](#).

---

## Section 2. UNIX incompatibilities

The reason why UNIX has issues when building applications is that not all UNIX distributions are the same, even though many are based on standard components and ideals. Understanding the effects of these differences is the first step toward building applications within different environments.

### Background

There are a number of different UNIX variants, including the free options, such as the various BSD (Berkeley Systems Division) variants (FreeBSD, OpenBSD, NetBSD) and Linux®. The differences stem, initially, from the source UNIX distribution used by a UNIX variant. Two source variants exist from AT&T (which ultimately evolved into what is now known as SVR4) and the version developed at the University of California Berkeley (which became known as UCB) and is today known as the BSD version.

Different companies have based their UNIX operating systems on one of these two basic editions and then added vendor-specific extensions and enhancements. Some companies, notably Sun, have changed their source for UNIX support. For example, Sun changed from a BSD core in SunOS to an SVR4 core in Solaris.

System V Release 4 is now the most common in commercial distributions and includes a core based on the original AT&T core with some additional BSD elements.

The result is that although the different UNIX versions (AIX®, HP-UX, Solaris, and others) are all technically UNIX, the differences between the systems mean that different libraries, header files, and even the tools used to build them mean that C source code cannot easily be migrated from one platform to another.

The actual functions have not changed; it is merely the location or definition of the functions that differ.

## POSIX compatibility

The POSIX group has worked very hard to try and standardize on a variety of different systems, including operating systems, utilities, and programming languages. The standards have a wide range and cover everything from the "standard" function calls and what they should return, to the capabilities and features of the OS on which those functions rely.

For UNIX, the primary POSIX standard of interest is 1003.1, which defines the interface between the application and the operating system.

POSIX standards have been adopted by a number of different companies in their operation systems, including Sun Microsystems, IBM, Digital, and Hewlett Packard. Even Microsoft provides a POSIX compatibility layer within Microsoft Windows.

The POSIX 1003.1 standard (otherwise known as POSIX.1) defines the function names used to execute specific operations within the OS, including the arguments, their format, and call order. The standard also specifies the expected return value and any errors (and includes standards for error numbers).

In short, any function call that is POSIX compliant should work on a variety of operating systems without any required modifications. For example, the `chdir()` function is defined in the POSIX standard, so the function: `int chdir(const char *path);` should be available on all UNIX variants that are POSIX compliant, and the argument value (a string), the return value (an integer), and the error messages raised in different environments are the same.

If the application that you are building is based on a specific UNIX variant, then migrating your system to POSIX standards is the first step toward making it more compatible, but it does not eliminate all the issues.

## Gauging the task at hand

Your first task is to identify the complexity of the application that you are trying to port. There are many different elements to a typical application. You need to determine which elements need to be redeveloped and which need work to make them more compatible across a range of platforms.

The elements you need to investigate are:

- Core C code -- The bulk of most C code is cross-compatible. Most incompatibilities in the C source code relate to the libraries and headers that the different elements rely on.

- Required libraries -- Different UNIX variants include a range of different libraries and tools, and some might even come with third-party libraries as standard.
- Build environment -- The Makefile is the standard build environment for most applications.
- Required build tools -- Some applications might need specific tools, such as `yacc`, `lex`, `rpcgen`, and others that will be needed within all the platforms you wish to build.

The bulk of most C code is of course cross compatible. Most modern C compilers are standards-based, so compiling the code is not the problem. Instead, the bulk of the issues when porting will be related to the libraries, headers, and limitations in the build environment.

---

## Section 3. Common areas of conflict

In this section, you'll examine some of the central issues around handling headers, libraries, build tools, and environment.

### Build tools and environment

The core elements to your porting process are the available build tools and environment. If you are using a standard Makefile, then the process is significantly easier, but you'll still need to take account of some critical differences.

For example, the available C compiler, C preprocessor, and tools like `yacc`, `lex`, and others need to be identified. You also need to take account of differences with the command line options and processes for different tools.

For example, most UNIX variants have a C compiler available as `cc` and a separate C preprocessor as `cpp`. However, some might prefer to run the preprocessor using the `-E` command line option to `cc`.

You can simplify the environment and process by making use of free software tools, such as `gcc`, `bison`, and `flex`, but again be aware of the limitations and differences.

The standard UNIX `yacc`, for example, creates files called `y.tab.c` and `y.tab.h` (when requested), but `bison` creates files based on the source name (for example,

generating `parser.tab.c` from `parser.y`). You need to account for the difference.

Also be aware that different tools are located in different places on different platforms. For example, Solaris includes `make` as standard (but not a C compiler), but it is located at `/usr/ccs/bin/make` rather than in the default path, such as `/usr/bin/make` as used on many other platforms.

The easiest way to account for these differences at a Makefile level (without using the GNU autotools) is to create a separate Makefile for each environment, altering each Makefile according to the platform. You can then use the appropriate Makefile when using `make`:

```
$ make -f Makefile.solaris
```

You still need to manage the individual differences between the files by hand, but the process should be much easier.

## Using the C preprocessor

Header files and most architectural and functional differences can generally be handled within C by using C preprocessor directives to select the different pieces of code.

The system works because C code is compiled before it is run through the C pre-processor (`cpp`). During compilation, the preprocessor looks for definition and uses a series of comparison macros to identify the existence (or otherwise) of a particular definition. You then specify the definition when building within a particular platform.

You might in fact already be using directives to enable debugging code. For example, you might specify the following definition when building an application within a Solaris x86 environment:

```
#define SOLARISX86
```

You can specify this definition either through a header file or by including the definition on the command line to the compiler:

```
$ cc -DSOLARISX86
```

Within the source code, you would then identify the correct source to use by checking for this definition. For example, in [Listing 1](#), make a special case for your

definition.

## Listing 1. Choosing an exception

```
#ifdef SOLARISX86
/* do solaris specific code */
#else
/* do other Unix code */
#endif
```

You can also use the `#ifndef` to check whether a directive has not been defined and use `#elif` to check additional directives during an `if...else...endif` section.

You can use the same system across your source code. For example, you can use it to include different header files ([Listing 2](#)) and account for the differences encountered earlier with the `localtime()` function.

## Listing 2. Choosing a header file

```
#ifdef SOLARISX86
#include <iso/time_iso.h>
#else
#include <time.h>
#endif
```

You need to use the same principles across your code, providing you know what you are looking for.

## Header differences

The most common problem when migrating between different UNIX variants is the header files used to define the structures, variable types, and functions that are available in the operating system.

Some header files have a different location. For example, the contents of `limits.h` are more or less standard across UNIX platforms, but there are located in different directories. Within AIX, for example, you import the header file using:

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

But within Solaris, you use:

```
#include <limits.h>
```

In some cases, the information you require is either not in the same file, differs from

the definition you have used on your original platform, or simply not available at all.

In the former situation, you need to find the location of the function definition, structure, or variable you are looking for. The best way is to use `grep` to search for the location, as shown in the following code, where you are looking for the `PIPE_MAX` definition on a Solaris host.

```
$ find /usr/include -exec grep -il PIPE_MAX {} \;  
/usr/include/sys/param.h  
/usr/include/limits.h
```

When the definition you are looking for does match the definition on the source platform, you can sometimes insert an alternative definition. Be careful, however, not to change the meaning of a system or library call; changing the forward function definition does alter the underlying library function.

If a definition does not exist, then it might indicate that the library or interface on which it also relies does not exist either.

## Library differences

Different UNIX variants place components into different libraries and might require the inclusion of a number of libraries that are not a requirement for a different platform. A classic example of this in action can be applied to the networking libraries. On many UNIX platforms, the necessary libraries for building network applications are automatically included in your applications when they are linked.

Within Solaris, however, you must specifically add these libraries for them to be linked with your application:

```
$ cc -o server server.c -lnsl -lsocket -lresolv
```

There is no easy way to list all of the different options and potential differences across the many different UNIX variants and functions. The only way you can resolve this issue is on an application-by-application basis. Try compiling the application and waiting for the missing symbol warnings.

For example, if you do not include the math library, you might get errors like those shown in [Listing 3](#).

### Listing 3. Missing symbol errors

```
calcparse.tab.o: In function `yyvsparse':  
calcparse.tab.c:(.text+0x53a): undefined reference to 'pow'
```

```
calcparse.tab.c: (.text+0x573): undefined reference to 'fmod'
calcparse.tab.c: (.text+0x5bb): undefined reference to 'log'
calcparse.tab.c: (.text+0x818): undefined reference to 'cos'
calcparse.tab.c: (.text+0x829): undefined reference to 'tan'
calcparse.tab.c: (.text+0x83c): undefined reference to 'cos'
calcparse.tab.c: (.text+0x849): undefined reference to 'tan'
calcparse.tab.c: (.text+0x856): undefined reference to 'asin'
calcparse.tab.c: (.text+0x863): undefined reference to 'acos'
calcparse.tab.c: (.text+0x870): undefined reference to 'atan'
collect2: ld returned 1 exit status
```

Once you have a list of missing symbols, you can look up the man page on the target platform and identify the required library.

For entirely missing functions, you might need to build the functions yourself or provide them from a third party library. For example, the GNU glibc includes many functions that might not be available in your native UNIX C library that you might be relying on.

When building the functions yourself, it is helpful to create a single C source file that contains the function definitions for different platforms, and then use the directive tricks shown earlier to select whether to build the functions on different platforms.

## Byte order differences

Beyond the basics of header files and libraries, you start to get into very platform-specific differences that are, in general, more difficult to resolve.

The most fundamental of these differences is the byte order of the host CPU. The byte order affects the way in which multi-byte data strings are referenced and accessed. This is because of the way the CPU operates.

Big endian CPUs (including SPARC, PA-RISC, and PowerPC) reference information by most significant byte first, while little endian CPUs (primarily Intel) reference by least significant byte first. This has the effect of completely reversing an address from, for example, ABCD to DCBA.

This does not affect strings, because strings are based on individual bytes, but it does affect multi-byte values, such as a 32-bit address, where the byte order would treat the value as different on big and little endian CPUs.

There is no simple method of resolving the issue, but there are ways in which you can reference and store information that does not rely on direct access to multi-byte data, so trigger the issue.

## Processes, signaling, and IPC

There are minor differences between the different `fork()` and threads implementations within different UNIX variants. Not all UNIX variants support all the `fork*()` functions, for example. IRIX supports the `m_fork()` function that is actually more akin to one of the various thread libraries. However, even with the realms of threads, different implementations and standards are not widespread.

Different UNIX variants support a different range of signals. Many of the core signals remain consistent; for example, `SIGHUP` has a value of 1, `SIGKILL` has a value of `NINE`, but don't rely on the availability of OS-specific signals, such as `SIGJVM1` (Solaris-specific), to be available on all platforms. Generally only the signals up to 15 are identical under most platforms. Beyond 15 (`SIGTERM`), signals are completely OS-specific.

Also be aware that Interprocess Communication (IPC) methods are not standardized. Although most UNIX variants support the SVR4 IPC, be aware there is not a blanket level of support. You should generally stick to more open IPC standards, such as named pipes.

---

## Section 4. Using the GNU autotools

The only issue with the approaches discussed thus far is that they require a significant amount of management to introduce and organize the different elements. They also require access to all of the different platforms that you need to support and the effects might increase your development time significantly to take into account all of the various options and alternatives.

### GNU autotools overview

The GNU autotools package is a system that produces a skeleton set of configuration rules and files. When the configuration script is run on a new target host, it examines the operating system and the requirements of the underlying source code and produces a suitable header configuration and build environment (based on the standard Makefile) that builds the application on the host.

If you have ever built an open source application on your machine, then you'll probably be aware of the following sequence:

```
$ ./configure
$ make
$ make install
```

The `configure` script is part of the distribution of the source code. It uses a series of configuration files to determine the various elements required to build. The main benefit of the autotools system is that it has the scripts ability to determine the configuration and availability of components on the host on which the script is being executed, compared to the required elements of the source that produce a working build environment.

The entire process works, because the autotools system relies on a combination of pre-known values (for example, it knows the target types, libraries, and header file availability) for specific platforms, combined with runtime-derived information that determines the availability of specific functions and header files.

You do not, for many applications, need to make changes to your code manually to make them compatible. Also, you do not even need to be aware of a target host for the `configure` script to work. I have developed applications that use the autotools system that have successfully compiled and built without any changes to the code on HP-UX, Solaris (SPARC and x86), Linux, Mac OS X, and BSD.

Obviously, to reap the benefits of the system, you must first instruct the autotools environment how you would normally build your application, and then use the autotools system to scan your source code and determine the functions, libraries, and other components required to make your application work.

## Setting the project structure

The first step toward using autotools is to create a suitable structure for your application. For this example, you'll be using a simple calculator application (see [Downloads](#)) that relies on a lexical analysis component (which requires `lex`) and a grammar component (which requires `yacc`), along with a couple of other files that support certain operations. You can see a list of the files in [Listing 4](#).

### Listing 4. The source files for a calculator

```
total 28
-rwxrwx--x 1 mc mcslp 458 Jan 25 17:19 Makefile*
-rwxrwx--x 1 mc mcslp 129 Apr 16 1997 calc.h*
-rwxrwx--x 1 mc mcslp 4216 Jun 11 1997 calcparse.y*
-rwxrwx--x 1 mc mcslp 136 Jun 8 1997 const.c*
-rwxrwx--x 1 mc mcslp 643 Apr 16 1997 fmath.c*
-rwxrwx--x 1 mc mcslp 1095 Jan 25 17:16 lex.l*
```

A hand written Makefile for building the calculator is in [Listing 5](#).

### Listing 5. A handwritten Makefile

```
YFLAGS = -d
```

```

PROGRAM      = calc
OBJS         = calcparse.tab.o lex.yy.o fmath.o const.o
SRCS         = calcparse.tab.c lex.yy.c fmath.c const.c
CC           = gcc      #C compiler

all:         $(PROGRAM)

.c.o:        $(SRCS)
             $(CC) -c $*.c -o $@ -O

calcparse.tab.c:      calcparse.y
                     bison $(YFLAGS) calcparse.y

lex.yy.c:           lex.l
                   flex lex.l

calc:              $(OBJS)
                   $(CC) $(OBJS) -o $@ -lfl -lm

clean:;           rm -f $(OBJS) core *~ \#* *.o $(PROGRAM) \
                   y.* lex.yy.* calcparse.tab.*

```

To set up the application for use with autotools, delete the Makefile, create a new directory (let's call it calc), and then a sub directory (src) into which you copy all of the source files. You can see the layout -- ready for autotools -- in [Listing 6](#).

### Listing 6. Sources files ready for building with autotools

```

./calc
./calc/src
./calc/src/calc.h
./calc/src/calcparser.c
./calc/src/calcparser.h
./calc/src/calcparser.y
./calc/src/const.c
./calc/src/fmath.c
./calc/src/lex.c
./calc/src/lex.l

```

You are now ready to start configuring the source with autotools.

## Creating the core Makefiles

You need to create a Makefile.am file within each directory of your source directory, including the root of the project. The Makefile.am in the root is used to reference the content in the other directories; the Makefile.am in each source directory is used to define the build process and requirements for the information in that directory alone.

So, for your example, the Makefile.am in calc will look like the following text:

```
SUBDIRS=src
```

This simply specifies the list of subdirectories that contain an environment that will be configured (and built) with the configure system.

The Makefile.am file within your main src directory looks like [Listing 7](#).

### Listing 7. The src/Makefile.am skeleton for building you default target

```
calcprgdir=../  
calcprg_PROGRAMS=calc  
calc_SOURCES=lex.l calcparser.y fmath.c const.c  
AM_YFLAGS=-d  
calc_LDADD=-lfl -lm
```

The prefix of the individual lines is important: the prefix is used to identify the target to which the option applies.

The first line in [Listing 7](#) specifies the ultimate target directory for the application when the `make install` is executed. While building this demonstration, you'll use the parent directory as the target install.

The second line specifies the name of the application that will be generated. The third lists the source files required to build the target. Note that although your application relies on C files that are automatically generated by separate tools (namely `lex` or `flex` and `yacc` or `bison`) you do not need to explicitly specify this stage. This is because the autotools package already knows how to build the C source from these source files, based on their extension.

The fourth line sets a general option to add the `-d` command line option to the `yacc` parser. Incidentally, the use of `yacc/bison` is yet another example of where the pre-configured knowledge of the autotools system comes into play. Autotools knows that UNIX platforms might have `yacc` installed, and if they don't have `yacc`, they have the GNU `yacc` tool `bison` installed. The two tools generate the same information, but in different files; for convenience, `bison` includes a `yacc` compatibility mode. When the `configure` script that autotools generates for your code is executed on a system, it will determine whether `yacc` or `bison` is available and adjust the command line for this stage of the process automatically.

The last line from [Listing 4](#) specifies the additional libraries required to build this target. Note that you only specify the library names as they would be specified to `cc`; the `configure` script determines where these libraries are located and whether it needs to add specific directories to the command when building the target.

With the basic build process configured, you need to get autotools to scan your source code and determine the functions and other elements in the source that

might have different sources and definitions within different environments.

## Scanning the source

The `configure.ac` file defines the configuration-specific elements of your source code. The contents of this file are used to generate the necessary configuration scripts that ultimately configure the build environment on a target host.

You do not have to create this file yourself; you can use the `autoscan` function to generate a skeleton version of this file for you automatically.

```
$ autoscan
autom4te-2.59: configure.ac: no such file or directory
autoscan-2.59: /usr/bin/autom4te-2.59 failed with exit status: 1
```

Don't worry about the error -- it's just indicating that it didn't have a `configure.ac` file on which to base the scan. It should create a file called `configure.scan` the first time the tool is run. You can see the contents of this file in [Listing 8](#).

### Listing 8. Default autoscan configuration

```
#                                     -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ(2.59)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([src/calc.h])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC

# Checks for libraries.
# FIXME: Replace `main' with a function in `-lfl':
AC_CHECK_LIB([l], [main])
# FIXME: Replace `main' with a function in `-lm':
AC_CHECK_LIB([m], [main])

# Checks for header files.
AC_FUNC_ALLOCA
AC_HEADER_STDC
AC_CHECK_HEADERS([stddef.h stdlib.h])

# Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST

# Checks for library functions.
AC_CHECK_FUNCS([pow])

AC_CONFIG_FILES([src/Makefile])
AC_OUTPUT
```

The `FIXME` elements give a good indication of what you need to edit. First, rename the file from `configure.scan` to `configure.ac`.

You need to reconfigure a number of elements here. Start with `AC_PREREQ` and change the values to reflect the name of the package, the version and an email address for contact.

Then add a line that configures the system for `automake`:

```
AM_INIT_AUTOMAKE
```

You also need to update the `AC_CHECK_LIB` lines to check for specific functions within the libraries you are using. For this application, you just need to make sure that you have the math library (`-lm`) available. Finally, you need to add two lines that include the configuration definition for running `lex` and `yacc`, and the files that rely on them.

The resulting file should look something like [Listing 9](#).

### Listing 9. The final `configure.ac`

```
AC_INIT(calc, 1.0, mc at mcslp.com)
AC_CONFIG_SRCDIR([src/calc.h])
AM_INIT_AUTOMAKE
AC_CONFIG_HEADERS([config.h])

# Checks for programs.
AC_PROG_CC
AM_PROG_LEX(src/lex.l)
AC_PROG_YACC(src/calc.y)

# Checks for libraries.
AC_CHECK_LIB([m], [sin])

# Checks for header files.
AC_FUNC_ALLOCA
AC_HEADER_STDC
AC_CHECK_HEADERS([stddef.h stdlib.h])

# Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST

# Checks for library functions.
AC_CHECK_FUNCS([pow])

AC_CONFIG_FILES([src/Makefile])
AC_OUTPUT
```

This file will be used by `autoconf` to determine the functions that need to be found to build the application.

## Modifying the source

The autotools system works by generating configuration files and, more importantly, a `config.h` header file that is used to contain all of the necessary definition elements

required to build the script.

You need to add the `config.h` header file to your C source files so that they load the configuration information during compilation.

You might also need to make some other minor changes to your code. For example, the configure scripts assume that a yacc file like `calc.y` will produce two files, `calc.h` and `calc.c`. Make sure this won't overwrite any existing, or simply change the name of the `yacc` source, so that the file generation won't overwrite your existing files.

## Creating the configure script

You now have a Makefile skeleton, and a configuration skeleton for the project. You need to turn this into four different components:

- A header file skeleton (`config.h.in`) that holds specific compatibility information for your project.
- The `aclocal.m4` file, a m4 macro that builds the necessary macros to convert your configuration file.
- `Makefile.in`, a template Makefile for each of your directories and sub directories.
- The configuration script (`configure`) that takes all of the above into consideration when determining the build environment and creates the necessary header files and Makefiles to build your application.

There are four steps to this process. First, generate the m4 macros:

```
$ aclocal
```

You can ignore any warnings generated by this command, but take notice of any errors; they probably indicate a typo in your `configure.ac`. Then create the template header file:

```
$ autoheader
```

You are now ready to generate the template Makefiles. Before you do this, you should create a number of files that are part of a standard autotools configuration. These files are:

- `COPYING` -- The rules for copying your source -- defaults to the GPL.

- AUTHORS -- A list of the authors.
- ChangeLog -- The history of changes to the source and releases.
- INSTALL -- The steps for installing the application.
- NEWS -- News for the application.
- README -- A simple README file about the application.

For the moment, you can just create empty files:

```
$ touch INSTALL NEWS README AUTHORS ChangeLog COPYING
```

Now you can create the Makefile templates:

```
$ automake -ac
```

The `-a` option tells `automake` to add missing files and the `-c` option copies those missing files into the directory.

Finally, you need to generate the configuration script that will actually perform the configuration determination on a target host:

```
$ autoconf
```

The entire sequence (and sample output) can be seen in [Listing 10](#).

### Listing 10. Configuration build sequence

```
$ aclocal
/usr/share/aclocal/pth.m4:43: warning: underquoted definition of _AC_PTH_ERROR
  run info '(automake)Extending aclocal'
  or see http://sources.redhat.com/automake/automake.html#Extending-aclocal
/usr/share/aclocal/pth.m4:55: warning: underquoted definition of _AC_PTH_VERBOSE
/usr/share/aclocal/pth.m4:61: warning: underquoted definition of AC_CHECK_PTH
/usr/share/aclocal/glib.m4:8: warning: underquoted definition of AM_PATH_GLIB
$ autoheader
$ automake -ac
$ autoconf
```

If you check your root directory, you should be able to identify the main components, such as the `configure` script, `config.h.in` and `Makefile.in`, as seen here in [Listing 11](#).

### Listing 11. Sample directory structure

```

-rw-rw-rw- 1 root mcslp      0 Jan 25 15:45 AUTHORS
-rw-r--r-- 1 root mcslp 18002 Jan 25 15:45 COPYING
-rw-rw-rw- 1 root mcslp      0 Jan 25 15:45 ChangeLog
-rw-r--r-- 1 root mcslp  9498 Jan 25 15:45 INSTALL
-rw-rw-rw- 1 root mcslp     12 Jan 25 15:44 Makefile.am
-rw-rw-rw- 1 root mcslp 18101 Jan 25 16:06 Makefile.in
-rw-rw-rw- 1 root mcslp      0 Jan 25 15:45 NEWS
-rw-rw-rw- 1 root mcslp      0 Jan 25 15:45 README
-rw-rw-rw- 1 root mcslp 31731 Jan 25 15:52 aclocal.m4
drwxr-xr-x 7 root mcslp     238 Jan 25 15:52 autom4te.cache/
-rw-rw-rw- 1 root mcslp  2563 Jan 25 15:52 config.h.in
-rw-rw-rw- 1 root mcslp  2634 Jan 25 15:51 config.h.in~
-rwxrwxrwx 1 root mcslp 174254 Jan 25 15:52 configure*
-rw-rw-rw- 1 root mcslp    690 Jan 25 15:52 configure.ac
-rw-rw-rw- 1 root mcslp    718 Jan 25 15:50 configure.ac~
-rwxr-xr-x 1 root mcslp 15936 Jan 25 15:45 depcomp*
-rwxr-xr-x 1 root mcslp  9233 Jan 25 15:45 install-sh*
-rwxr-xr-x 1 root mcslp 11014 Jan 25 15:45 missing*
drwxrwxrwx 16 root mcslp    544 Jan 25 16:08 src/

```

You can now try to configure your application.

## Trying the new configuration

To try the configuration, just run `configure`. You can see the script determining the different library and header files required in [Listing 12](#).

### Listing 12. A sample configuration script execution

```

$ ./configure
checking for a BSD-compatible install... /bin/install -c
checking whether build environment is sane... yes
/bin/sh: /mnt/mc/Active: No such file or directory
configure: WARNING: `missing' script is too old or missing
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking for flex... flex
checking for yywrap in -lfl... yes
checking lex output file root... lex.yy
checking whether yytext is a pointer... yes
checking for bison... bison -y
checking for sin in -lm... yes
checking how to run the C preprocessor... gcc -E
checking for egrep... grep -E
checking for working alloca.h... yes
checking for alloca... yes
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes

```

```

checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking stddef.h usability... yes
checking stddef.h presence... yes
checking for stddef.h... yes
checking for stdlib.h... (cached) yes
checking for an ANSI C-conforming const... yes
checking for pow... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating config.h
config.status: executing depfiles commands

```

You can then run `make` to build your application (see [Listing 13](#)).

### Listing 13. Running make

```

$ make
make all-recursive
make[1]: Entering directory `/export/data/calc'
Making all in src
make[2]: Entering directory `/export/data/calc/src'
bison -y -d calcparser.y
flex lex.l
if gcc -DHAVE_CONFIG_H -I. -I. -I..
-g -O2 -MT fmath.o -MD -MP -MF ".deps/fmath.Tpo" -c -o fmath.o fmath.c; \
then mv -f ".deps/fmath.Tpo" ".deps/fmath.Po"; else rm -f ".deps/fmath.Tpo";
exit 1; fi
sed '/^#/ s|lex.yy\.c|lex.c|' lex.yy.c >lex.c
rm -f lex.yy.c
if gcc -DHAVE_CONFIG_H -I. -I. -I..
-g -O2 -MT tmath.o -MD -MP -MF ".deps/tmath.Tpo" -c -o tmath.o tmath.c; \
then mv -f ".deps/tmath.Tpo" ".deps/tmath.Po"; else rm -f ".deps/tmath.Tpo";
exit 1; fi
if gcc -DHAVE_CONFIG_H -I. -I. -I..
-g -O2 -MT const.o -MD -MP -MF ".deps/const.Tpo" -c -o const.o const.c; \
then mv -f ".deps/const.Tpo" ".deps/const.Po"; else rm -f ".deps/const.Tpo";
exit 1; fi
if test -f y.tab.h; then \
to=`echo "calcparser_H" | sed \
-e 'y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/' \
-e 's/[^ABCDEFGHIJKLMNOPQRSTUVWXYZ]/_/g'`; \
sed -e "/^#/!b" -e "s/Y_TAB_H/$to/g" -e "s|y\.tab\.h|calcparser.h|" \
y.tab.h >calcparser.ht; \
rm -f y.tab.h; \
if cmp -s calcparser.ht calcparser.h; then \
rm -f calcparser.ht ;\
else \
mv calcparser.ht calcparser.h; \
fi; \
fi
if test -f y.output; then \
mv y.output calcparser.output; \
fi
sed '/^#/ s|y\.tab\.c|calcparser.c|' y.tab.c
>calcparser.ct && mv calcparser.ct calcparser.c
if gcc -DHAVE_CONFIG_H -I. -I. -I..
-g -O2 -MT lex.o -MD -MP -MF ".deps/lex.Tpo" -c -o lex.o lex.c; \
then mv -f ".deps/lex.Tpo" ".deps/lex.Po"; else rm -f ".deps/lex.Tpo"; exit 1; fi

```

```
rm -f y.tab.c
if gcc -DHAVE_CONFIG_H -I. -I. -I..
-g -O2 -MT calcparser.o -MD -MP -MF ".deps/calcparser.Tpo"
-c -o calcparser.o calcparser.c; \
then mv -f ".deps/calcparser.Tpo" ".deps/calcparser.Po";
else rm -f ".deps/calcparser.Tpo"; exit 1; fi
gcc -g -O2 -o calc calcparser.o lex.o fmath.o tmath.o const.o -lfl -lm -lm
make[2]: Leaving directory `/export/data/calc/src'
make[2]: Entering directory `/export/data/calc'
make[2]: Nothing to be done for `all-am'.
make[2]: Leaving directory `/export/data/calc'
make[1]: Leaving directory `/export/data/calc'
You can try out the application just to make it has been built properly:
$ ./src/calc
45*69
3105
```

You can revert to an unbuilt and unconfigured state within your build directory by using the `distclean` target:

```
$ make distclean
```

By sharing your files over an NFS connection and using this method to "reset" after each sample build, you can easily test on multiple platforms without having to copy the distribution to each host.

## Limitations of the autotools

The autotools system takes out a lot of the complexity of building multi-platform compatible applications, but it is not an ultimate solution.

The autotools solution cannot take into account special libraries, third-party or your own, that are not part of the distribution file (although there is nothing stopping you adding them, or creating specific library autotool configurations).

The autotools also cannot account for any OS-specific elements that you have used. If the library or function that you are relying on is unique to Solaris (for example, if it is unique to the Solaris kernel interface), then autotools will be unable to resolve that.

The autotools system only resolves the major issues of UNIX and C library differences, their header relationships, and the build environment required to compile the link the elements. It cannot simulate an alternative library and kernel environment for the purpose of porting software.

This does not mean that autotools should not be used; it is probably the most effective build environment tool for porting -- but do not expect it to work miracles. To get the best out of autotools, you must be willing and prepared to combine it with some of the techniques discussed earlier in this tutorial in order to ensure cross

compatibility.

Finally, no matter how much work you do, there is no substitute for actually having your target platform available to test on. Autotools might simplify the build process, but don't expect it to eliminate the need for debugging and testing.

---

## Section 5. Summary

Developing applications that work under multiple UNIX platforms requires an understanding of the underlying issues that affect the compilation and linking process. For nearly all situations, the problems relate to two distinct areas, the header files that support standard system and kernel functionality and the libraries used to support specific extensions and functionality. Hidden behind these differences are the complex issues of specific operating system limitations. For example, differences in the signals supported by different UNIX variants might cause problems.

Some of these differences can be alleviated by using tricks to get around the differences and issues. To simplify the header, library, and build environment issues, you can make use of the autotools/autoconf package produced by GNU. This uses a combination of known differences and a discovery script to determine the environment on a target during the build process, and then builds a suitable build script and header file to account for the platform-specific issues.

While autotools can simplify the build process, it is not designed to account for all of the differences between platforms. It can't, for example, account for missing functions, libraries, and core OS differences, but it will make a significant difference.

## Downloads

| Description                       | Name                  | Size  | Download method      |
|-----------------------------------|-----------------------|-------|----------------------|
| Calc source for autotools example | mcb-calculator.tar.gz | 120KB | <a href="#">HTTP</a> |

[Information about download methods](#)

# Resources

## Learn

- [Porting enterprise applications from UNIX to Linux](#) is a guide for porting from UNIX to Linux, but provides a lot of useful information about porting and differences in general.
- [Guide to porting from Solaris to Linux on x86](#) provides specific Solaris information for porting applications.
- [The Technical guide for porting applications from Solaris to Linux](#) gives more detailed technical information about the Solaris operating system.
- Although targeted at the zSeries® platform, in an z/OS® UNIX platform, there is a lot of good general and practical advice that can be used for all UNIX porting projects in the [z/OS UNIX System Services Porting Guide](#).
- [The IBM AIX Porting Redbook](#) contains specific AIX porting information and technical details.
- Want more? The developerWorks [eServer](#) zone hosts hundreds of informative articles and introductory, intermediate, and advanced tutorials on the eServer brand.
- The IBM developerWorks team hosts hundreds of [technical briefings](#) around the world which you can attend at no charge.

## Discuss

- Participate in the [eServer™ forums](#).
- [developerWorks blogs](#): Get involved in the developerWorks community.

## About the author

### Martin Brown

Martin C. Brown is a former IT director with experience in cross-platform integration. A keen developer, he has produced dynamic sites for blue-chip customers, and is the technical director of Foodware.net. Now a freelance writer and consultant, MC, as he is better known, works closely with Microsoft as an SME, is the LAMP Technologies Editor for *LinuxWorld* magazine, is a core member of the AnswerSquad.com team, and has written books such as *XML Processing with Perl, Python and PHP*, and the *Microsoft IIS 6 Delta Guide*. MC can be contacted at [questions@mcspl.com](mailto:questions@mcspl.com).