SPP - Low Tech, Practical, UNIX Software Portability

John Sellens – jmsellens@uwaterloo.ca
University of Waterloo, Waterloo, Ontario, N2L 3G1 Canada

Abstract

The great number of UNIX variants and the huge amount of good, freely available software makes simple software portability a very worthwhile and useful goal. Portability has traditionally been obtained with software package specific solutions, configuration scripts, `#ifdef`'s and so on. This paper describes SPP, a small collection of include files, library routines and utilities that can be installed once and utilized by many, different software packages. SPP does not require that a software author change his or her programming style or techniques, and frees the author from worrying about the variations among machines, allowing him or her to concentrate on the program instead of the operating system.

1 The Motivation for SPP

At the University of Waterloo, most UNIX software support is provided by central support groups, and much effort is expended to provide a consistent set of software and tools across all supported platforms[1]. Automated tools are used to build and distribute software to multiple hardware/operating system combinations (16 at last count) from a single source repository. The source must be buildable without human intervention, and any source configuration or option selection must be done automatically.

Hundreds of megabytes of source is used to build hundreds of megabytes of installed software, much of it obtained over the network, or from other freely-distributable sources. Much of this software is excellent, and many packages come with extensive configuration files and scripts, that work hard to make software installation as painless as possible[1].

If portability to multiple architectures is desired (and it usually is), software authors must go to considerable effort to construct configuration scripts, or resort to `Makefile` editing and *cpp* `#ifdef`'s to achieve a measure of portability. The amount of effort required to construct "portable" software can sometimes discourage software authors from distributing their software. In addition, the portability scheme adopted can sometimes make it more difficult to port it to another architecture[2].

There does not seem to be a common set of tools that a software author can easily use to write portable software. Each author must make the effort to make the software portable, and must learn enough about enough systems to do it properly. Much effort is expended (and duplicated) in the goal of portability (or even transportability). If software can be written by the original author in a way that anticipates portability, and using a relatively common set of tools, then everyone will benefit[3]. And, with an appropriate set of tools, an author can create software that is far more likely to build and install on platforms that the author has no knowledge of.

SPP is intended to provide a small set of tools that are a practical approach to portability. The tool set must be small and simple enough that it won't be a burden to install and use, and it must be easy to understand how it works, and what it will do (e.g. the output of *make install* must be understandable). SPP can only be

[1] The most notable is probably Larry Wall's *Configure* script; the version distributed with *perl* 4.0 is over 4,000 lines long, and can generate 500 lines of output! *Configure* was, however, generated by another program, *metaconfig*.

[2] Consider the case of code liberally sprinkled with "`#ifdef vax`" to determine byte-ordering.

[3] This is, after all, part of the "UNIX philosophy".

useful if it is widely installed and available; more software will be more easily installable if the author can rely on the availability of a useful portability tool set.

The name, SPP, is intended to reflect this reality, and these goals. The name is an abbreviation for "Standard Portability Package", though the "S" is also intended to stand for software, small, simple, and sufficient. The "S" also stands for "Practical", but the name "PPP" was already taken.

The approach described here has been used (in various variations and extremes) over a number of years at the University of Waterloo. SPP is an attempt to collect what we've learned in a form that can be used outside the university, and without any extraneous ornamentation.

2 Why Standards Aren't (Yet) Enough

In a perfect world, we would all have compilers that adhere to the appropriate standards, we would all have libraries that adhere to the appropriate standards, we would never want to do anything vendor-specific, wouldn't need to work-around hardware idiosyncrasies, and all vendor-provided software would be completely up to date, and, of course, completely bug free.

But, of course, this is not likely to be the case in very many non-trivial environments. There are many systems running with old software that may never be upgraded, OS vendors implement standards at varying rates of speed, and different bugs appear on different platforms. It is necessary to be able to work around incomplete or malfunctioning libraries and command sets, different versions of relevant standards, and vendors that are slow (or fast) in responding to change. SPP is intended to provide a way to work around these kinds of problems.

With luck, the need for SPP will lessen as time goes by, and it will get smaller and smaller, and eventually it will be obsolete. In the meantime, it can be a useful tool for software portability.

3 The Approach Used by SPP

SPP is a small collection of three types of objects:

1. C language include files
2. a small library of C language routines (`libspp.a`)
3. a small number of utility programs, typically implemented as Bourne shell scripts

The SPP include files provide a number of basic abilities. They allow a program to determine the architecture and version of a machine, in case the program needs to do something different on different machines. They allow indications of common differences between UNIX variants; for example the return type of `readdir()` is `struct direct *` on some machines and is `stuct dirent *` on some other machines. They allow the definition of standard macros and types that may be missing on "older" machines. And they allow for the indication of certain machine attributes that are usually external to the C language interface, such as the name of the kernel, or the contents or location of various files (such as the `utmp` file).

For example, on a Sun machine running SunOS 4.1.3, `SPP_OS_SUNOS_4_1_3` is defined (via the include file `<spp/config.h>`). `SPP_OS_SUNOS_4_1_3` is used as the key to define other flag macros that indicate attributes of SunOS 4.1.3 (via `<spp/attrib.h>`), such as `SPP_HAS_UT_HOST`, which indicates that `struct utmp` has a `ut_host` field on this machine. (All macros defined by SPP include files start with `SPP_` to avoid potential conflicts with other software.) All the appropriate SPP include files are included as a result of including the file `<spp/spp.h>`.

The *spp* library provides a number of "common" routines that are not provided with the operating system on some machines. For example, the standard `strdup()` function is missing on some machines, so the *spp* library provides a version of it. The *spp* library on a given machine contains only those routines that are needed to supplement that particular architecture, and so its contents vary from machine to machine, and it might even be empty on some very "standard" machines. The library isn't intended to contain absolutely every routine that might be missing on a machine – it is intended to provide only relatively common, standard, routines that are likely to be useful in a variety of situations.[4]

The SPP utility programs are typically only those required to build and install software, and are usually simple cover scripts to cover variations across different architectures. For example, the *sppranlib* command shields the software author from having to know which machines have (or need) a *ranlib* command, or from having to write complicated tests in a `Makefile`.

Underlying the SPP implementation is the idea that things should be kept as small and as simple as possible, and that extraneous material should be kept to a minimum. With the basic `SPP_OS_*` definitions and enough knowledge, a software author can write portable code. The other parts of SPP, such as the attribute definitions and the library routines, are merely convenience features to make it easier to write software that will work on machines that the author has no access to.

One major benefit of the SPP implementation is that a software author (or someone trying to port existing software) doesn't have to use all of SPP; the programmer can use only those parts that are needed (or wanted) in a particular situation. For example, making existing software portable is sometimes only a matter of changing the original `Makefile` to use the *sppinstall* command to install the compiled program.

4 What SPP Doesn't Do

SPP was created to help deal with a certain set of problems, and it isn't intended to solve every portability problem that exists. For starters, it is very UNIX oriented. While a number of the concepts would port easily to other environments, it assumes an underlying tool set and filename syntax that is typically found only on UNIX systems.

SPP is oriented towards C language programming, and won't solve all the portability problems that exist when writing in other languages, such as the UNIX shells, or *perl*. Some of the utility programs (such as *sppinstall*) can be useful in many situations, but the include files and the library routines are probably of use only to the C language programmer.

SPP doesn't try to solve every portability problem, just the most common ones. For example, it doesn't solve the problems inherent in trying to write software that will work on a variety of operating systems. But it does provide the basic building blocks that should allow a software author to make extensions that would help in solving less common problems.

SPP probably isn't the portability solution for large, complicated systems. For example, SPP probably isn't suited for use with the X Window System.

5 Makefiles, Install Scripts, and So On

SPP provides a small set of tools for use in building and installing software. Typically implemented as simple shell scripts, these tools are intended to augment the standard UNIX tools in simple but useful ways. The SPP tools are *sppmake*, *sppinstall*, and *sppranlib*.

sppmake is a simple wrapper for *make* that preprocesses an `SPPMakefile` and invokes *make* on the resultant `Makefile`. This is similar to *imake* (and was, in fact, inspired by it), but with two minor philosophical differences. In most instances, *imake* is used with a separate "template" file that often defines a large number of rules and macros for building the software. While this is useful in many cases, especially with large systems, it usually means that one can't tell what *imake* will do by just looking at an `Imakefile`. *sppmake* is typically used to implement a small number of simple conditionals in an otherwise normal

[4] The *spp* library is similar to the GNU "libiberty" library, but the latter includes additional, non-standard functions. The *spp* library is expected to shrink over time, and it seems likely that the libiberty library will grow.

`Makefile`, e.g. to select different compiler options on different architectures. This means that, typically, *sppmake* is easier to start using than *imake*, since the `SPPMakefile` is more likely to resemble a normal `Makefile`. The other difference is that *imake* typically uses pre-defined *cpp* symbols in order to determine the current machine's architecture; *sppmake* typically uses an architecture-specific include file that defines appropriate SPP-specific symbols.

sppinstall is a "standard" install command, that allows the use of the same options and command name on multiple machines. For example, BSD-based machines have an *install* command, DEC OSF/1 machines have an *installbsd* command, IRIX machines have *bsdinst.sh*, and AIX machines have two different *install* commands, one in `/usr/bin` and one in `/usr/ucb`. *sppinstall* is just a version of the old, standard BSD *install* command that frees the software author from having to worry about what command name and arguments to use to get a program installed.

sppranlib serves a similar purpose. Some machines require the *ranlib* command to be used when creating libraries, some don't, and some don't even have a *ranlib* command. *sppranlib* looks for a *ranlib* command, runs it if it finds one, and does nothing otherwise. Again, this just makes it easier for a software author (or porter) to make things work on multiple machines.

These three simple tools seem to provide most of the functionality that is needed when building software on multiple, different machines. *sppmake* provides a simple way to customize the build process for different machines, and *sppranlib* and *sppinstall* provide useful utilities to make things easier for a software author.

6 Alternative Approaches to Portability

The problems that SPP attempts to address have been faced by many people and projects, and there have been many different approaches to solving (or avoiding) these problems. While SPP is not for everyone, it provides a simple approach with a low "start-up cost" i.e. it's easy to obtain and install, and it's simple to use. However, a comparison with alternate portability approaches may be useful.

One simple approach is for a software author to provide the necessary "missing" functionality. For example, since some machines do not include a `strdup()` function, an author could write a version of `strdup()` and include it with the software. An extension of this would be a custom support library for use in building the software. SPP essentially does this already, and the advantage of using SPP in this case is that the results can be shared by many projects, authors, and machines. SPP also frees an author from worrying about differences between machines, so that the author's effort can be directed to the program itself, rather than to the deficiencies that exist in some environments.

Another, more involved, approach is the "if your tools don't work, get different tools" approach. The portability problems that occur due to different libraries and commands can be solved by replacing the offending libraries and commands. For example, an author could adopt the GNU tool set, *gcc*, *make*, *install*, and the GNU C library, and many of the basic problems would be solved. An author would still need to deal with machine-specific differences (like the `utmp` file format and location), but the tools and library routines would be consistent. This approach doesn't address the problem of avoiding bugs on different systems, but with "identical" libraries, this is less likely to be a problem. This "replacement" approach (like SPP and like most others) requires that the replacement tool set be available anywhere an author wants to be able to build and install the software, and, if the software is to be distributed, this can place a major additional burden on the users of the software. SPP faces this same problem, but installing SPP is (intentionally) a smaller task than installing a complete set of replacement tools.

There are configuration tools, like GNU *autoconf* and Larry Wall's *metaconfig*, that provide a way to automatically create configuration scripts that will "research" a machine and generate an appropriate `Makefile` and other configuration information. These tools are very useful for larger projects (like *gcc* and *perl* for example), but they appear to be overkill for many smaller projects. There are many worthwhile and useful programs that require only a small amount of portability assistance, and for which the learning curve and configuration requirements of this kind of tool might be daunting for even an experienced software author. Another disadvantage to this kind of tool is that they can create `Makefiles` that are incomprehensible to many people – it is usually valuable to be able to type *make -n install* and be able to understand what it is about to do to a machine.

One other approach to portability is that used in the X Window System source distribution; using *imake*, large template files, and the `<X11/Xos.h>` include file. `<X11/Xos.h>`, which tries to isolate many of the machine-specific differences in one common file, has much in common with parts of SPP, but it tends to be X-specific, rather than oriented towards general portability problems (and there is no guarantee that `<X11/Xos.h>` won't change in the next release). The X approach, in essence, created a large, self-contained, X-specific environment for building and installing programs, which is probably appropriate for a system of that size. But it did result in the build and configuration tools being hard to understand, and the average user or installer is unlikely to take the time to fully understand these tools. It also resulted in a system that is not trivial to adapt to other projects, and so *imake* has tended to be used only with X-related software.

The primary advantage of SPP is that it is simple, easy to learn and use, and that, by and large, it allows software authors to continue to use the standard UNIX tools in familiar ways, without having to make large-scale changes in order to deal with (sometimes) minor portability problems.

7 SPP Implementation

The SPP implementation has deliberately been kept as simple as possible, and can be installed using only the tools that should already be on every UNIX system.

The SPP include files are kept in a separate `spp` directory, in order to avoid name clashes with other files. There are currently 5 main SPP include files:

- `<spp/config.h>` is the file that causes the correct `SPP_OS_` macros to be defined for the current machine. This file is usually modified by each installation to include the appropriate sub-file. For example, on a machine running SunOS version 4.1.2, `<spp/config.h>` will include the file `<spp/config/sunos/412.h>`.
- `<spp/attrib.h>` uses the `SPP_OS_` macros to determine which "attribute" macros to define, e.g. `SPP_HAS_UT_HOST` is defined on most machines, but not on those that don't have a "host" field in the `utmp` file.
- `<spp/types.h>` provides additional standard typedefs, like `time_t`, if they are missing on particular machines.
- `<spp/libspp.h>` declares the routines in the *spp* library, since if they are missing on a machine, then they are unlikely to be declared in the standard include files. The routines are declared only if they are included in the *spp* library on a particular machine, in order to avoid overruling or clashing with the real (correct) declarations.
- `<spp/spp.h>` is a cover file that includes all the others, and should normally be the file included in a program.

All names defined by SPP, with the exception of standard library routines provided as part of the *spp* library, start with `SPP_` to avoid name clashes with other software.

The `<spp/config.h>` file causes the macros `SPP_OS_MAJOR` and `SPP_OS_MINOR` (and, when appropriate, `SPP_OS_TEENY`) to be defined to indicate the version of the operating system in use, as well as specific macros naming the OS. For example, on a machine running SunOS version 4.1.2, the following would be defined:

```
#define SPP_OS_MAJOR    4
#define SPP_OS_MINOR    1
#define SPP_OS_TEENY    2
```

```
#define SPP_OS_SUNOS
#define SPP_OS_SUNOS_4
#define SPP_OS_SUNOS_4_1
#define SPP_OS_SUNOS_4_1_2
```

The OS version is defined in two different ways, making it easy, for example, to test for a particular version

```
#if defined(SPP_OS_SUNOS_4_1_2)
    /* do something special on SunOS 4.1.2 */
#endif
```

or to test for a version later than a particular version

```
#if defined(SPP_OS_SUNOS) && (SPP_OS_MAJOR >= 5)
    /* Oh, oh - things are very different now ... */
#endif
```

Note that `<spp/spp.h>` will likely cause C language declarations to be included, so it shouldn't be included by an `SPPMakefile` – `<spp/config.h>` should be used in an `SPPMakefile`.

The utility programs, *sppmake*, *sppinstall*, and *sppranlib*, are implemented as Bourne shell scripts, and are installed by simply copying them into the destination directory. The are deliberately kept as simple as possible. For example, the current version of *sppranlib* is

```
#!/bin/sh

if [ -f /usr/bin/ranlib ]; then
    exec /usr/bin/ranlib ${1+"$@"}
fi

exit 0
```

although the test might have to be extended to handle alternate locations for the *ranlib* command.

sppmake takes an `SPPMakefile` as input, runs it through *cpp*, puts the output into a temporary file called `#sppmaketmp` (for reference in case of problems when running *make*), and invokes *make* on that makefile.

The *spp* library is probably the most complicated part of SPP, because there is actual C code involved, and a slightly complex `SPPMakefile` that tries to include routines in the library only when that routine is missing from the standard libraries on a machine. Most of the routines in the *spp* library have been plundered from other freely-distributable libraries.

It is important for the SPP source to build and install as easily as possible, and for it to be almost completely self-contained, because if it gets too complicated, it will no longer be doing what it was intended to do.

8 Hints on Source Code Portability

Writing portable software isn't always an easy task. Even if we ignore different window systems and user interfaces, and non-UNIX operating systems, it still isn't an easy task. A software author needs to be aware of the applicable standards (like ANSI C for example), and has to be aware of potential portability pitfalls, usually before starting to write the software.

The C preprocessor and the `#if` and `#ifdef` conditionals are a major advantage when trying to write portable code. But, conversely, the misuse of `#if`'s and `#ifdef`'s can actually impede portability. For example, there is a common tendency to do things like

```
#if defined(sun) || defined(vax) || defined(sequent)
    /* blah blah blah */
#else
    /* some other blah blah blah */
#endif
```

and to have that kind of construct sprinkled throughout the code. The result is that porting to a new machine requires going through all of the code, searching for places to add yet another identifier to the appropriate `#if`'s.

Another common problem is the assumption that a `long` or an `int` is a particular size, or that bytes or bits appear in a particular order. Now that 64-bit processors (like DEC's Alpha chip) are becoming more common, there is starting to be a sizeable amount of code that looks something like this

```
#ifdef alpha
    int value;
#else
    long value;
#endif
```

And, historically, byte-ordering has often been determined by something like this

```
#ifdef vax
    /* little endian */
    ...
#else
    /* big endian */
    ...
#endif
```

which has proven to be a bad way of doing things.

The definitions in SPP's include files are, in part, an attempt to promote "healthy" coding styles. The attributes defined in `<spp/attrib.h>` encourage the use of conditional tests based on the attributes of a machine, rather than on the set of machines that happen to have that attribute. For example, this code

```
#if !defined(mips) && !defined(irix)
    strncpy( ut->ut_host, hostname, sizeof(ut->ut_host) );
#endif
```

is harder to maintain and port to new machines than this code

```
#ifdef SPP_HAS_UT_HOST
    strncpy( ut->ut_host, hostname, sizeof(ut->ut_host) );
#endif
```

especially if the `utmp` file entries are manipulated at multiple places in the code.

The problem of what type of variable to use is addressed in part by SPP's `<spp/types.h>` file. Historically, we've all known that the `time()` function returned a `long`, but using the newer, standard, `time_t` return value instead means that the code is more likely to run correctly on a 64-bit machine, other future machine types, or with future versions of `time()`. `<spp/types.h>` should ensure that all "standard" types are available to software authors on every machine, and, with luck, will serve as a gentle reminder to authors to define their own types, in their own header files.

Another important technique that leads to more portable software is the use of include files to isolate machine-specific tests and differences. If software has `#ifdef SPP_OS_SUNOS` in more than one file, or in the middle of a bunch of code, it is probably an indication of a portability problem waiting to happen. If all machine dependencies and machine-specific `typedef`'s are isolated in one file (with a fairly obvious name, like `attrib.h` or `machines.h`), then porting to a new machine (or a new OS version) will usually require changing only that one file.

Outside of the program code itself, there are portability problems (and understandability problems too), often found hiding in a package's `Makefile`. The most common problem is the need to specify machine-specific options in the `Makefile` itself, which means it's impossible to compile source on multiple platforms without manual intervention. Another problem is `Makefile` commands that are in fact miniature shell scripts, usually in an attempt to cope with inconsistencies in OS vendors' command sets. *sppmake* provides a solution for the former problem, and small utilities like *sppranlib* and *sppinstall* provide ways to deal with the latter. The result is a `Makefile` that is far more understandable, more maintainable, and more portable.

9 Potential Extensions to SPP

It seems likely that SPP will be (should be) extended to provide additional attributes and library routines as it gets used in more situations. There may be a tendency to want to provide for every possible attribute or machine difference, but there is a danger that adding too much to SPP will obscure the "simple" part of SPP, and make it hard to use. Each addition to SPP will have to be weighed against this possibility.

It is possible that, as more people use SPP, it will become obvious that SPP is missing some important functionality, or that it "missed the boat" in some important area. Experience suggests that the relatively small set of tools that SPP provides will be sufficient for a large number of small to medium size programs, and so the expectation is that SPP will still be useful even if there are a number of situations where SPP isn't useful.

As SPP is extended, software packages may be built that require more recent versions of SPP. SPP defines `SPP_VERSION` and `SPP_RELEASE` to allow authors to check for particular versions of SPP in their software.

10 Conclusions

SPP provides a reasonable approach to portability, based on some relatively simple files and tools. It provides an easy way to write portable software, or to port existing software to new machines or environments.

SPP isn't necessarily (and probably isn't) the first place any of these portability problems have been addressed, and it probably doesn't introduce any radical new approaches. What it does attempt to do is to bring together a small collection of useful (and, hopefully, sufficient) tools that can solve many of the most common portability problems, in a way that can be adopted by many authors and installations.

However, SPP is not the right tool to solve all portability problems. For example, it is probably not suited for very large pieces of software, such as *gcc* or the X Window System distribution. It provides a tool set and basic approach for software authors; if SPP itself doesn't suit the needs of a particular project, it should provide useful examples and approaches to solving particular portability problems.

11 Acknowledgements and How to Obtain SPP

SPP is based on tools and ideas developed over the years at the University of Waterloo. SPP is a somewhat formalized, documented, and non-Waterloo-specific collection of the software and ideas. Accordingly, credit for the good ideas in SPP belongs to the UNIX software support staff at the University of Waterloo, and particularly the "software guys" in the Mathematics Faculty Computing Facility.

The SPP distribution is available, via anonymous FTP, from the machine `math.uwaterloo.ca` in the file `pub/spp/spp.shar.Z`. Contact the author (at `jmsellens@uwaterloo.ca`) if other arrangements are necessary.

The current version of SPP defines a relatively small, but common, set of machines. It is expected that this set will grow as other sites install SPP and provide appropriate information about other machines and architectures.

A Installing SPP

Installation of SPP has deliberately been kept simple, and uses only the lowest common denominator tools that should be on every UNIX machine.

The steps for installing SPP are:

1. Extract the source from the archive into an appropriate directory.
2. Change `spp/config.h` so that the appropriate machine and OS-specific configuration file is included on your system(s). If you have only one machine, simply `#include` the appropriate configuration file. If you have multiple machines of different types or versions, you may either have a separate `spp/config.h` file for each type of machine, or you may `#ifdef` as necessary to suit your installation and environment.
3. Change `Makefile` to set the destination locations of the SPP components. The `Makefile` suggests typical locations; you will want the commands to end up in your path, and the include files and libraries to end up where your compiler can find them, hopefully without extra *-I* or *-L* options.
4. Type the command *make install* and SPP should be ready to use.

If you have a system that is not currently covered by SPP, please contact the author to arrange for its inclusion. After an initial phase where many machines are added, SPP will, with a little luck, become usable on almost any UNIX system with very little customization needed.

B Examples of Use

Because SPP has been kept simple with independent components, some simple examples should be all that is necessary to demonstrate some typical uses of SPP.

An example of a simple `SPPMakefile` is:

```
#include <spp/config.h>

BINDIR = /usr/local/bin

CFLAGS = -O
#if defined(SPP_OS_AIX)
/**/# we want the BSD-ish version of things on AIX
CFLAGS = -O -D_BSD
#endif

all: prog

/**/# we use -lspp to pick up any missing "standard" routines
prog: prog.o
        $(CC) $(CFLAGS) -o prog prog.o -lspp

install: prog
        sppinstall -c -s -m 755 prog $(BINDIR)/prog
```

```
clean:
        rm -f *.o prog core
```

Note that the `SPPMakefile` includes `<spp/config.h>` to avoid getting the C language code that would be included by `<spp/spp.h>`.

An example of a program fragment that makes use of an SPP attribute is:

```
#include <stdio.h>
#include <utmp.h>
#include <time.h>
#include <spp/spp.h>

setutentry( ut )
struct utmp *ut;
{
    strncpy( ut->ut_line, "unknown", sizeof(ut->ut_line) );
    strncpy( ut->ut_name, "unknown", sizeof(ut->ut_name) );
#ifdef SPP_HAS_UT_HOST
    strncpy( ut->ut_host, "unknown", sizeof(ut->ut_host) );
#endif
    ut->ut_time = time( (time_t *)0 );
}
```

Note that this code includes `<spp/spp.h>` to get all the SPP-related include files, rather than just the configuration information required by the `SPPMakefile`.

References

[1] Sellens, John, "Software Maintenance in a Campus Environment: The Xhier Approach", *LISA V Proceedings*, September 1991, pp 21-28.