iMatix Corporation sprl

13/15 rue des Ateliers 1080 Bruxelles, Belgium
Tel.+32-75-235-984, fax.+32-2-414-8404
http://www.imatix.com
info@imatix.com

Technical White Paper

Software Portability

iMatix Corporation makes software that runs on most computer systems. We do this using a straight-forward and autonomous method. In this paper we describe our method for constructing truly portable programs that run on most current and future computer systems.

Copyright

Version Information

Written: 4 February 1999
Revised: 23 January 2000

Disclaimer

Defining Portability

Portability means that the same software will run on many different computer systems. Many programs are 'portable', but the value of this claim depends largely on how that is achieved.

A truly portable program can be moved to a new, unexpected computer system with a minimum of effort, and without impact on existing use.

The Portability Domain

A software program, or application, runs in a certain domain. Some examples are:

- A web application runs on some server, and talks to the user using an HTML page.
- A mainframe application runs on a mainframe and talks to the user using a 'dumb' terminal screen.
- A windows program runs on a PC and talks to the user using the Windows GUI.
- A command-line program runs on a PC or a Unix system and talks to the user using a simple command and response model.

In general, an application is designed for one such domain, and usually stays there. Some applications move from one domain to another, but this usually implies a complete restructuring.

A portable application is at least portable within one domain, and possibly within multiple domains as well. It is possible, with foresight, to construct a single application that will work with all the domains noted above.

Portability usually means "will run on most major computer systems that are capable of supporting this type of application or domain".

Types of Portability

Let's look at some examples of portable and so-called 'portable' applications:

- The **zip** tool was ported in 1990 and later to a large range of systems, including IBM mainframes and Amiga systems. The zip tools were ported by adding system-dependent code to the application source code. Some routines, such as file access functions, were isolated into separate modules that were then rewritten for each platform. I call this technique 'Portability by Evolution'. Most 'portable' software is ported like this.
- The **Perl** programming language runs on all Unix systems and was ported to Windows some years ago. The Windows version of Perl has 'forked' from the principal version so that it has become a separate product, sharing some core code. I call this technique 'Portability by Forking'. This technique is a refinement of the evolutionary

approach, usually needed when the evolutionary changes are too large maintain a single product.

- The Java programming language offers a platform for portable applications. Any program written in Java is portable to any computer system that supports Java. I call this 'Portability by Decree'.
- Our Xitami web server runs on most computer systems. It was never 'ported', and the source code for Xitami does not contain any system-dependent code. All portability issues were handled by a separate layer, the SFL library. A similar technique is used to make the Linux operating system portable to many different CPUs. I call this 'Portability by Design'. Obviously this is the approach we prefer and recommend.

Portability by Evolution

Instructions: take an application and make it run on a new system by changing its code until it runs. Use conditional instructions to use the new code only on the new system, and test the application on other systems to see it still compiles and works there.

Advantages: anyone can port such an application.

Disadvantages: the application source code becomes more and more complex as more and more system-dependent code is mixed with the application's base code. It becomes hard to understand and eventually becomes impossible to improve or maintain.

Lesson: sprinkling system-dependencies throughout an application's base code are a bad way to port it. This approach will eventually kill-off the application.

Portability by Forking

Instructions: take an application and make it run on a new system by changing its code until it runs. Do not worry about how the application runs on other systems.

Advantages: anyone can port such an application.

Disadvantages: each time there is a new release, someone has to struggle to include the new base code into the ported / forked version. Eventually this becomes too much effort, and the forked version starts its own evolution, or stagnates.

Lesson: forking an application is a poor solution in any case, since the effort required to maintain it doubles. The net result is that the same original application ends-up in the hands of multiple teams of developers. This happened with Unix in the 1980's.

Portability by Decree

Instructions: define a virtual machine on which a set of applications can run, and then ask companies to implement this virtual machine without adding or omitting anything. There are many ways to implement such a virtual machine – Java uses a interpretative approach. Standards like POSIX define a virtual machine implemented as a runtime environment for compiled programs.

Advantages: in principle, all the nasty portability issues caused by historical accident can be ignored.

Disadvantages: it is very hard to define a new realistic virtual machine from scratch. Inevitably there will be false starts, new versions, and new releases, often incompatible with previous releases. This is the case with the Java virtual machine. There is no way to guarantee that competitors will respect the virtual machine - indeed they will often work deliberately to create incompatible virtual machine, the so-called extended subsets.

Lesson: trying to impose portability by decree is very difficult. The example of Java shows that this model is easily broken or corrupted by competitors who do not share the same interests in creating a standard virtual machine. The example of POSIX shows that it takes many years for vendors to provide support for such standards, and they do this grudgingly and often only when faced with the alternative of extinction.

Portability by Design

Instructions: define a virtual machine on which a set of applications can run, and then implement this machine yourself on each target system. A full interpretative approach is not realistic, but designing a virtual machine as a runtime environment is quite straight-forward.

Advantages: in principle, all the nasty portability issued cased by historical accident can be covered-up and hidden. In practice this can be quite a delicate task. The cost of extending the virtual machine to work on a new computer system is quite small (typically a few hours to a few days).

Disadvantages: the virtual machine restricts somewhat the domain of any application using it. If the virtual machine is badly-designed, it may not work on future unexpected computer systems. Badly-designed virtual machines will also slow-down applications.

Lessons: this is the best approach, if done right.

Comparing The Approaches

Looking at the approaches above, we can see that different projects have taken different routes to creating portable applications. Some routes work better than others. Some approaches are not tenable in the long run.

The Economics of Portable Code

Looking at the economics of building portable code, it is incredible that the vast majority of software development is still done without any regard at all to this.

The cost case for a portable approach is easy: portable applications are cheaper, better, and longer-lasting than non-portable ones.

Portable Applications are Cheaper

All operating systems change, and many have completely disappeared in the last two decades since the computer business reached the desktop.

A typical non-portable application has a half-life of perhaps one to two years. After this time, half of the application's functionality will be broken by changes in the operating system. So, the application developers need to spend a significant effort simply to keep the application working, disregarding any effort spent on improving it.

In a portable application, there is still work to keep up with operating system changes. However, since system-dependent code is isolated in one layer, this is easy to change. Further, improvements and corrections benefit all applications that use the portability layer.

In my experience, a portable application is up to 90% cheaper than non-portable applications, over extended periods.

Portable Applications are Better

It follows from the argument that portable applications need less routine maintenance that the developers have more time to spend on real improvements.

Since portable applications last longer (see next point), improvements are accumulated.

A bogus counter-argument to this is that since portable applications last longer, they accumulate more and more faults until they become completely unmaintainable. The argument is bogus because any programmer good enough to appreciate and use a portability layer (or even to develop one) is capable of improving code over time, rather than degrading it.

Portable Applications Last Longer

In typical non-portable applications, changes in the OS eventually destabilise the application to the extent that it becomes cheaper to rewrite it than to keep maintaining it. Think of the millions of programs written for MS-DOS or Windows 3.1. Portable code is aloof from such changes, and can survive migration from MS-DOS to Windows 95 as easily as to Unix. Portability also means 'portable into the future'.

Three Steps to Portability

These is, then, our recipe for constructing a portable application using a long-term, economic approach:

1. Rather than solving the specific problem for one application, solve the problem for the entire application domain. This is always a good tactic in any design, great in any software design, and essential (in our eyes) for portability issues.
2. Define a virtual machine that supports the needs of your domain. This is easiest if you already have experience working in this domain on different computers systems.
3. The virtual machine supports the principal non-portable requirements of the application. It may also support important requirements that are fully portable but which are so commonly used that it's worth answering them once and for all.

Then, we construct a simple virtual machine from those functions. For a typical command-line program like a web server, this requires support for file access, TCP/IP sockets, process creation, etc.

We've done this, and the resulting Open Source library is called 'SFL' (Standard Function Library).

The iMatix SFL Library

Our basic design divides each application into two layers: a "technical layer" and a "functional layer." The technical layer, reused in all applications, encapsulates all non-portable features, and provides a set of useful library functions. The functional layer is portable, and constitutes the real "application."

Our technical layer is a library of C functions that we called the ***Standard Function Library*** (SFL). SFL is further subdivided into different packages. For instance, the string package provides various string manipulation functions, the socket I/O package provides a set of functions to access Internet sockets, the date package provides a set of date conversion functions, and so on.

Abstracting Functions

The SFL provides a series of abstractions for functions that we need on all systems, but that must be implemented with non-portable, system-specific code. It also encapsulates functionality, as does any useful library. We generally combine these two needs.

For example, the process control package, sflproc.c, provides functions to create, monitor, and kill background processes. Under UNIX we create a child process by using the fork() and exec() system calls. Under 32-bit Windows we use the CreateProcess() system call. But the implementation is not trivial: we need quite a lot of supporting code to redirect input and output streams, to ensure that the child process is correctly started, and to handle errors.

Our abstraction for process_create() looks like this:

```
PROCESS                         /*  Returns a PROCESS token             */
process_create                                                           (
    char *filename,            /*  Name of file to execute             */
    char *argv [],              /*  Arguments for process, or NULL      */
    char *workdir,             /*  Working directory, or NULL          */
    char *std_in,              /*  Stdin device, or NULL               */
    char *std_out,             /*  Stdout device, or NULL              */
    char *std_err,             /*  Stderr device, or NULL              */
    char *envv [],              /*  Environment variables, or NULL     */
    Bool  wait                 /*  Wait for process to end             */
)
```

If you abstract functions correctly, you don't lose performance or functionality. Of course, the "correct" way is not always obvious, and sometimes takes a few iterations to discover. Often, when faced with various options, we choose something closest to the UNIX model. In the last few years, many platforms are moving towards Posix compatibility. One consequence of this is that the Posix, UNIX-like abstraction is the most stable and long-term. The test of a good portability abstraction is that it does not change when it's ported to a new platform. Hindsight, experience, and access to a lot of documentation helps a lot here.

One well-understood benefit of packaged functionality is that the internals - what we call the technical layer - can be improved as needed without affecting the calling programs. For an SFL package, this "improvement" can mean porting to a new platform, or improving the way it works on a specific platform. When Ewen McNeill <ewen@imatix.com> ported the SFL to OS/2, he relied on the fact that OS/2 with EMX provides many UNIX-like functions. So, it was quite easy to make the process control package work on OS/2. Native OS/2 system calls provide better performance, but we're not obliged to make an optimal solution right away.

Case Study - a Portable Web Server

A project like the SFL needs several real, demanding client applications to test the technology and prove that it works. We decided to use the SFL in all our software development, starting with our web server, Xitami. In an Internet server program -- which has little or no user-interface -- the main portability issues are file handling, process control, and socket I/O.

The SFL file-handling package sflfile.c hides the differences between UNIX, VMS, and MS-DOS text file formats and filename syntaxes. For instance, the file_where() function searches along a path for some file. This is a simple concept, very useful to locate an applications' data files, but works quite differently on VMS, UNIX, or MS-DOS. The file_is_executable() function checks the file's attributes under UNIX, but actively searches for an .exe, .com, or .bat file under MS-DOS or OS/2. There are many issues that we do not cover (e.g. file locking); however such package such as the sflfile.c provides a framework for adding new functions as we need them.

The socket I/O package, sflsock.c shows how to avoid portability problems by a combination of simplification and stubborn audacity. The socket abstraction is itself widespread and pretty standard, being based on the BSD socket library. A simple socket-

based program is easily portable to all BSD-derived systems: UNIX, OS/2, and Digital OpenVMS. When we looked at using socket-based programs under Windows, however, we found two problems.

Firstly, the Windows socket library (Winsock) has a call interface that only partially resembles the BSD socket interface. Secondly, under Windows, sockets are usually connected to the user-interface code -- socket events being handled in the same way as mouse and keyboard events. This means that a Windows socket program is constructed totally differently from a BSD-style socket program.

Our preferred abstraction for sockets (as for anything else) is a set of simplified functions (connect, read, write,...) that both encapsulate the differences between systems, and add scaffolding code such as error handling. The question was: would such an abstraction work under Windows? Winsock provides the select() system call, which lets us collect socket events in Windows as we do in UNIX. So we basically cheat, and answer the portability question by avoiding Window's normal event-driven socket handling altogether. Surprisingly, perhaps, this works just as efficiently as the 'native' Windows approach.

The SFL does not address portability in user-interfaces. We're basically interested in programs that do little or no user input/output. When we compiled the web server, using Microsoft Visual C/C++ 4.0, as a console program, it worked much like its UNIX counterpart -- that is, simply and quickly. However, we wanted to build a Windows user-interface for the web server, mainly for marketing reasons. A common criticism of a 'portable' approach is that it restricts one to producing simplistic or functionally-poor applications. The Windows version of Xitami proves that this is not true; indeed, that a portable approach can give better results, and faster, than the system-specific approach.

We wrote a Windows user-interface program as two threads. The first thread manages a user-interface dialogue, and is not portable (we wrote different versions for 16-bit and 32-bit Windows, and for the Windows NT service version). The second thread runs like a classic UNIX daemon, in the background. The two threads communicate via shared variables.

It took about a week to build this front-end. The result is application that looks and feels like a native Windows application, but where the bulk of the application code is portable, and kept cleanly separate from the non-portable user-interface code.

To demonstrate the effectiveness of this approach: Xitami now runs on Windows 3.x as a 16-bit program, on Windows 95 and NT as a 32-bit program, on Windows NT as a service, and on Windows 95 and NT as a 32-bit DOS program. In all cases the bulk of the code remains unchanged, and well-isolated from the thin UI layer required to glue it to the specific Windows version. In contrast, native Windows web servers are highly version-specific. For instance, Microsoft's own IIS runs only on NT 4.0. Yet, Xitami is faster than IIS, and a relatively small program (the complete executable code is about 400 Kbytes).

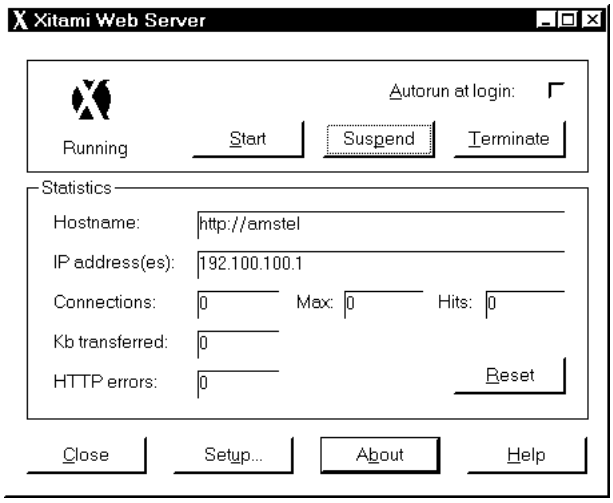

Figure 1: The Xitami Control Panel

Portability and the C Language

So far, we've discussed techniques that can apply to any generally-portable programming language -- many of the principles were developed by Leif Svalgaard <leif@ibm.net> and others in the 1980's for supporting portable COBOL applications. The C language presents its own peculiar challenges; mainly non-portable header files and library functions, and non-portable language constructs and data types.

Let's look at header files. The principle is that you add an `#include` statement for the various functions that you need. The problem is that on different platforms, these files (except for standard ANSI header files) sit in different directories, have different names, or may not even correspond at all.

For example, on most UNIX systems to use socket functions you must include the files <sys/socket.h>, <netinet/in.h>, and <arpa/inet.h>. Under IBM AIX you also need <sys/select.h>. Under Digital VMS you need only <socket.h> and <in.h>. Under Windows, you need <winsock.h>, and under OS/2 with EMX, you need <sys/socket.h>, <sys/select.h>, <netinet/in.h>, and <arpa/inet.h>.

At some point, this requires conditional macros (e.g. "#if __TURBOC__"). The question for us was how to do this with the least effort. Listing one shows how a careful programmer might include the socket header files in a program.

Different compilers predefine macros like WIN32, __i386__, __vax__, or _hpux. In each source program that uses socket functions or socket data types, our careful programmer could repeat these conditional macros. Eventually our programmer would see that copying code like this was a bad idea, and might create a single header file to handle the various includes. Of course, the same scenario applies to other system functions - files, directories, processes,... So, with some more work, our programmer could build a set of header files, each covering some set of functions.

This fictitious set of header files now poses its own maintenance burden. One day our programmer decides to compile the code on a new platform, or using a new release of the

C compiler. There are compile errors - each of the header files has to be changed to take into account new predefined macros, header file locations, and other changes.

Looking at this scenario, we saw a lot of unnecessary hard work. Our solution was two-fold. To start with, we decided never to use compiler-specific macros like __i386__. Instead, we would define a stable, clean set of macros (__UNIX__, __VMS__, __OS2__, __MSDOS__,...) using whatever internal mechanisms necessary. Our careful programmer can write conditional code ("#if __OS2__"), but when we want to handle a new compiler, there is at most one file to change. Secondly, we would bite the bullet and put all #include statements in one place. Again, the aim was to have a single point of focus for non-portable header files.

We wrote a single header file, prelude.h, that handles both these needs. Listing two shows the way that system-specific header files are included in prelude.h. This is a controversial approach: we had some heated discussions in comp.lang.c about its (de)merits. The decision about exactly which header files to include is not evident. (We take most, but not all, ANSI header files. Then we take those files needed to support sockets, directory access, timers, etc.) A program that uses the prelude.h will compile slower (two to five times slower) than a program that includes only those files it really needs. The prelude.h file makes a raft of definitions that may conflict with those the programmer wants to make. The final complaint is that we remove a level of control that many C programmers are used to exercising.

However, we consider this as an inevitable and worthwhile compromise along the way to true portability, as well as a good way to simplify an otherwise complex problem. We've used the prelude.h file in many projects, with or without the SFL, and it works well.

Another portability gotcha in C is the size of integer datatypes. On most systems an 'int' is 32 bits, but on some it's 16, and on others 64. Here we took a commonly-used approach, which is to define datatypes 'byte', 'dbyte', and 'qbyte'. These are always one byte, two bytes, and four bytes long. This approach probably rules-out systems with exotic words sizes, a fair compromise for us. Again, we put these type definitions into prelude.h, so that all programs would share them.

A final problem with writing portable C code is that the programmer must be careful to avoid non-portable library functions and data types. This is a matter of experience, good books, and a good compiler help function. Most compiler help systems will indicate whether a function is ANSI, POSIX, or system-specific. We assumed that all ANSI functions are supported on our target systems. We then rewrote common but non-standard functions such as strlwr(). We also assumed that most platforms will move towards POSIX compatibility, so we use POSIX constructs where possible. Largely, this approach works, though we find that it is important to regularly 're-port' applications to target platforms to ensure that non-portable constructs do not creep in.

Portable code built this way can be simple but functional. Listing three shows a portable directory listing program. This is a typical example of functionality ('directories') that exists on many platforms, which is useful in applications, but which requires non-portable code to use.

Redefining The Makefile

When we moved our C programs to various platforms, we found that compilers are not standard, not even on different versions of the same operating system. The C compiler is generally called 'cc', but often uses different arguments for optimisation, enforcing ANSI syntax, linking, etc.

The approach used in many multiplatform (but not portable) products is to build these differences into many makefiles, one per platform. Alternatively, to build the differences into a single multiplatform makefile. In any case, the user must choose a target system by issuing a command like 'make aix'.

To rebuild a package like the SFL, we have to issue a number of compile commands, create an archive file for the compiled programs, and link some executable programs. We decided that writing and maintaining traditional UNIX makefiles was too much work, and only a partial solution. There are tools that generate platform-specific makefiles: for example the **imake** tool used in the X window system. **imake** is powerful, but too complex for our needs, and too specific to UNIX platforms. Our needs were simple: recompile a set of C programs, build some object libraries, link some executables.

Taking a cue from the CERN libwww reference library installation, we wrote a 'c' script that detects the UNIX platform, and runs the appropriate commands to compile or link a program. The 'c' script fully abstracts the compiler interface on UNIX platforms. The result of this is that the programmer is more portable, as well as the software.

We additionally wrote a small tool that generates build scripts from a single portable command file. The tool (**otto**) generates scripts for UNIX (using the 'c' script), VMS, OS/2, and several MS-DOS compilers, and is easy to extend for other platforms. An **otto** script can test for required files, compile and link programs, copy, rename, and delete files, and run system-specific commands. To make life easier for the developer, **otto** also develops standard makefiles.

C and UNIX Portability Standards

In 1983, the ANSI C X3J11 Committee started the process of defining a standard C language that would run on all platforms, from embedded micro-controllers to supercomputers. ANSI C was published in 1990 as ANSI X3.159-1989, and was later replaced by the ISO standard ISO/IEC 9899:1990. The most significant difference between the earlier de-facto standard, K&R C, and ANSI C was a standard C run-time library with corresponding header files and function prototypes. Most (but not all) modern C compilers support ANSI C fully. One notable exception is SunOS, where the default compiler is K&R, and an ANSI C compiler must be separately purchased. A new ANSI standard is expected in 1999.

System V.4 UNIX is (or rather, was, until AT&T sold the UNIX trademark) the de-facto "standard" commercial UNIX, and provides "portability by inclusion": it combines various UNIX offshoots: System V.3.2, 4.3BSD, SunOS, XENIX. IBM's AIX, HP's HP/UX, and Sun's SunOS are modified versions of System V.4 or the earlier System V.3. After many years of industry conflict over the "UNIX" trademark and copyrights, Novell received the

UNIX trademark from AT&T, and then passed it to X/Open, who started to define a "standard UNIX".

OSF/1 is a consortium standard for UNIX from the Open Systems Foundation, started by a few large vendors as reaction to the UNIX policies adopted by AT&T. Digital's UNIX systems now use OSF/1, and IBM has stated its intention to replace their AIX operating system (largely based on 4.3BSD) by OSF/1. OSF/1 is principally System V.4 compatible, and also tries to support POSIX.1. Like System V.4 it often provides two or more alternative versions of system functions.

In 1995, X/Open and OSF were brought together under the Open Group, finally providing the basis for a single, standard UNIX.

POSIX -- unlike the commercial UNIX variations -- is an official standard from the IEEE. POSIX is actually a set of standards covering operating systems, programming languages, and tools. The POSIX.1 standard (IEEE 1003.1-1990) covers operating systems and looks much like a subset of UNIX. Non-UNIX operating systems (such as Digital's OpenVMS, Windows NT, and even IBM MVS) can and probably will eventually be POSIX compliant.

The promise is this: use only POSIX functions and your applications will be portable to all POSIX-compliant operating systems. In reality, POSIX.1 standardises a number of important system interfaces, but is not rich enough to provide the basis for many real-life applications. In an example of remarkably bad marketing, the IEEE does not make its standards documents freely available; these must be bought. The web site http://www.posix.com/ provides more information on obtaining IEEE standards documents.

Conclusions

The test of a portability toolkit is moving to a new system. In January 1997, Ewen McNeill <ewen@imatix.com> ported the SFL to OS/2. Ewen made changes for OS/2 to four SFL packages: the user-ID package, the socket package, the directory package, and the process-control package. Ewen also added OS/2 support to the 'c' script and Otto. In February 1997, Vance Shipley <vances@motivity.ca>, ported the SFL to SCO UNIXWare and SCO OpenServer 5.0, changing the 'c' script, the prelude.h file, and the process-control package. Again, the changes were minor, quickly made and tested. In both cases, as we expected, all our applications built on the SFL - including Xitami - ran without modification. Well, not exactly, because OS/2 showed-up a couple of dormant bugs.

Portability need not be a constraint on the development process. Rather, it is a complexity-reduction method that can help the developer deliver high-quality, stable, and efficient applications at a lower cost. Portability applies to software, to processes, and to people.

We built our portability toolkit by:

- identifying the types of application we wanted to support;
- identifying the types of target systems we wanted to support;

- finding an abstraction (an API) to protect the programmer from the differences in these systems;
- writing a function library to support this abstraction;
- writing the tools to support this abstraction.

For Further Reading

"Internetworking With TCP/IP Volume III: Client-Server Programming And Applications BSD Socket Version" by Douglas E. Comer and David L. Stevens, published 1993 by Prentice-Hall Inc. ISBN 0-13-020272-X. This is the bible on writing internet servers.

"Porting UNIX Software", by Greg Lehey, published 1995 by O'Reilly & Associates, Inc. ISBN 1-56592-126-7. An excellent book that helped us avoid most of the known UNIX portability pitfalls.

"Software Portability with imake", by Paul DuBois, published 1993 by O'Reilly & Associates, Inc. ISBN 1-56592-055-4. Provides a good description of the imake tool and its possibilities.

Source Listings

Listing one: how ***not*** to use socket header files

```
/*  Listing one: how *NOT* to use socket header files                      */

/*  Include files for Windows                                              */
#if defined WIN32 || defined _WIN32 || defined WINDOWS || defined _WINDOWS
#   include <windows.h>
#   include <winsock.h>

/*  Include files for OS/2                                                 */
#elif defined __EMX__ && defined __i386__
#   include <sys/socket.h>
#   include <sys/select.h>
#   include <sys/time.h>
#   include <sys/stat.h>
#   include <sys/ioctl.h>
#   include <sys/file.h>
#   include <sys/wait.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>

/*  Include files for Digital OpenVMS                                      */
#elif defined VMS || defined __VMS || defined __vax__
#   include <socket.h>
#   include <in.h>

/*  Include files for UNIX, except AIX                                     */
#elif defined unix || defined __unix__ || defined __hpux || defined SUN
#   include <sys/socket.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>

/*  Include files for IBM AIX                                              */
#elif defined _AIX || defined AIX
#   include <sys/socket.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>
#   include <sys/select.h>
```

```
#endif
```

Listing two: an extract from the Universal Header File

```
/*  Listing two: an extract from the Universal Header File                  */

#if (defined WIN32 || defined (_WIN32))
#   undef __WINDOWS__
#   define __WINDOWS__
#   undef __MSDOS__
#   define __MSDOS__
#endif

/*  __OS2__     Triggered by __EMX__ define and __i386__ define to avoid    */
/*              manual definition (eg, makefile) even though __EMX__ and    */
/*              __i386__ can be used on a MSDOS machine as well.  Here      */
/*              the same work is required at present.                       */
#if (defined (__EMX__) && defined (__i386__))
#   undef __OS2__
#   define __OS2__
#endif

#if (defined (__hpux))
#   define __UTYPE_HPUX
#   define __UNIX__
#   define _INCLUDE_HPUX_SOURCE
#   define _INCLUDE_XOPEN_SOURCE
#   define _INCLUDE_POSIX_SOURCE
#elif (defined (_AIX) || defined (AIX))
#   define __UTYPE_IBMAIX
#   define __UNIX__
#elif (defined (linux))
#   define __UTYPE_LINUX
#   define __UNIX__
# … etc
#elif (defined __UNIX__)
#   define __UTYPE_GENERIC
#endif


/*- Standard ANSI include files ---------------------------------------------*/

#ifdef __cplusplus                      /*  PA 96/05/29                     */
#include <iostream.h>                   /*  A bit of support for C++        */
#endif

#include <ctype.h>
#include <limits.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <float.h>
#include <math.h>
#include <signal.h>
#include <setjmp.h>


/*- System-specific include files -------------------------------------------*/

#if (defined (__MSDOS__))
#   if (defined (__WINDOWS__))          /*  When __WINDOWS__ is defined,    */
#       include <windows.h>            /*    so is __MSDOS__               */
#       include <winsock.h>            /*  May cause trouble on VC 1.x     */
#   endif
#   if (defined (__TURBOC__))
#       include <dir.h>
#   endif
#   include <dos.h>
#   include <io.h>
#   include <fcntl.h>
#   include <malloc.h>
```

```
#   include <sys\types.h>
#   include <sys\stat.h>
#endif

#if (defined (__UNIX__))
#   include <fcntl.h>
#   include <netdb.h>
#   include <unistd.h>
#   include <dirent.h>
#   include <pwd.h>
#   include <grp.h>
#   include <sys/types.h>
#   include <sys/param.h>
#   include <sys/socket.h>
#   include <sys/time.h>
#   include <sys/stat.h>
#   include <sys/ioctl.h>
#   include <sys/file.h>
#   include <sys/wait.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>
/*  Specific #include's for UNIX varieties                                  */
#   if (defined (__UTYPE_IBMAIX))
#       include <sys/select.h>
#   endif
#endif

#if (defined (__VMS__))
#   if (!defined (vaxc))
#       include <fcntl.h>              /*  Not provided by Vax C           */
#   endif
#   include <netdb.h>
#   include <unixio.h>
#   include <types.h>
#   include <socket.h>
#   include <dirent.h>
#   include <time.h>
#   include <pwd.h>
#   include <stat.h>
#   include <in.h>
#endif

#if (defined (__OS2__))
/*  Include list for OS/2 updated by EDM 96/12/31
 *  NOTE: sys/types.h must go near the top of the list because some of the
 *        definitions in other include files rely on types defined there.
 */
#   include <sys/types.h>
#   include <fcntl.h>
#   include <malloc.h>
#   include <netdb.h>
#   include <unistd.h>
#   include <dirent.h>
#   include <pwd.h>
#   include <grp.h>
#   include <sys/param.h>
#   include <sys/socket.h>
#   include <sys/select.h>
#   include <sys/time.h>
#   include <sys/stat.h>
#   include <sys/ioctl.h>
#   include <sys/file.h>
#   include <sys/wait.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>
#endif
```

Listing three: directory list program

```
/*
 *  Name:       testdir.c
 *  Title:      Test program for directory functions
 *  Package:    Standard Function Library (SFL)
 *
 *  Written:    96/04/02 <sfl@imatix.com>
 *  Revised:    96/12/12 <sfl@imatix.com>
```

```
 *
 *  Synopsis:   Testdir runs the specified or current directory through
 *              the open_dir and read_dir functions, formatting the output
 *              using format_dir.
 *
 *  Copyright:  Copyright (c) 1991-1998 iMatix Corporation
 *  License:    This is free software; you can redistribute it and/or modify
 *              it under the terms of the SFL License Agreement as provided
 *              in the file LICENSE.TXT.  This software is distributed in
 *              the hope that it will be useful, but without any warranty.
 */

#include "sfl.h"

void handle_signal (int the_signal)
{
    exit (EXIT_FAILURE);
}

int main (int argc, char *argv [])
{
    NODE    *file_list;
    FILEINFO *file_info;
    char     *sort_type = NULL;

    signal (SIGINT,  handle_signal);
    signal (SIGSEGV, handle_signal);
    signal (SIGTERM, handle_signal);
    if (argc > 2)
        sort_type = argv[2];

    file_list = load_dir_list (argv [1], sort_type);
    if (file_list)
      {
        for (file_info  = file_list-> next;
             file_info != (FILEINFO *) file_list;
             file_info  = file_info-> next
            )
            puts (format_dir (&file_info-> dir, TRUE));
        free_dir_list (file_list);
      }
    /*  Check that all allocated memory was released  */
    mem_assert ();
    return (EXIT_SUCCESS);
}
```