# A Platform Independent Source Code Engineering System

*Martin Neath*
*neath@itg.ti.com*

Texas Instruments Incorporated
Information Technology Group
Austin, TX

*ABSTRACT*

The Platform Independent Source Code Engineering System (PISCES) is an easy to use source code revision control, configuration, and automated make file system for use in large software projects. PISCES is a collection of several public domain utilities modified to work closely together to provide the software engineer with tools to simplify the process of rapidly porting and maintaining a C/C++ source code base across heterogenous hardware platforms and software environments. This paper describes PISCES, presents a software methodology development model for structuring and maintaining a source code base, discusses the major components and how they have been modified to closely cooperate with each other, and provides an introduction to its use through several annotated examples.

## 1. Introduction

The Platform Independent Source Code Engineering System (PISCES) is a collection of utilities providing a platform-independent source code revision control, configuration, and automated make file system. It is designed to be used by engineers on large software projects to control the complexities of source file dependencies and make file systems across heterogenous hardware platforms and operating system environments. PISCES is built upon the Revision Control System (RCS), the macro-make file facility (imake), an automatic file dependency list generator (mkdepend), the DECUS ANSI C preprocessor (cpp), and the GNU file differentiator program (diff).

The immediate impetus for creating PISCES was to ease the process of porting and maintaining a large software project written in C and C++ running on SPARC work stations, a MIPS Ultrix file server, a large VAX VMS system, and PS/2 (TM) personal computers running either SCO UNIX® or IBM® OS/2 (TM) Extended Edition. Our software methodology requirements established the need for a revision control system with support for versioning and strict file locking, a make facility to control the build procedure that gives us a high degree of confidence in file dependency detection and automatic updates, and an easy means for recursively performing maintenance tasks and rebuilding and running regression test suites. We examined the native tools available on each target platform and found varying levels of support for each of these tasks. No single platform, however, provided all the desired functionality and no two systems were consistent in either the programmer or command line interface.

As a result, we developed PISCES to allow the programmer to have a consistent interface to a common suite of tools providing powerful source code manipulation and configuration functions across very different platforms. These tools are all written in C and "glued" together through a single software methodology with an expected behavior utilizing machine independent macros and machine-specific command names and arguments. The

---

PS/2 is a trademark of International Business Machines Corporation.
UNIX is a registered trademark of AT&T.
IBM is a registered trademark of International Business Machines Corporation.
OS/2 is a trademark of International Business Machines Corporation.

modifications made to the public domain utilities are largely centered around removing any operating system bias towards UNIX, completely parameterizing all pathname determination and construction, and allowing for the joint development of both C and C++ object files in a single environment.

PISCES does not have infinite flexibility or the capability to solve very complex configuration problems. Rather, it is a simple-to-use collection of tools ideal for the 90% of configuration and build operations centered around construction of C and C++ object files, libraries, and programs on diverse hardware and software platforms. This paper describes PISCES, presents a software methodology development model for structuring and maintaining a source code base, briefly discusses the major components and the modifications made to each, examines the supported rules and actions for program development, provides an introduction to its use through several annotated examples, and lists the requirements and procedure for adding support for a new hardware platform or operating system.

## 2. Supported Hardware Platforms, Operating Systems, and Compilers

In order to be a useful and valueable tool, PISCES must run on several major hardware platforms and software environments. Since our immediate goal was to enable the development and configuration of a large C and C++ project on several different machines, we identified the following six hardware platforms and associated operating systems and language facilities as the minimal required environments on which PISCES should be available and supported:

- SUN SPARC supporting SunOS(TM) C and the AT&T C++ translator
- DEC VAX/VMS supporting VAX C and the Oasys C++ language system
- DEC MIPS/Ultrix supporting Ultrix C and the AT&T C++ translator
- TI S1500 supporting GNU C and the AT&T C++ translator
- IBM PS/2 running OS/2 EE with IBM C/2(TM) or Microsoft® C, and Glockenspiel C++
- Intel 386/486 platforms supporting SCO UNIX/C and Glockenspiel C++

The various UNIX environments are fairly similar and represent relatively easy platforms to support. The VMS and OS/2 systems, however, are quite different in nature and organization and, in places, were difficult to support. Since portability and identical functionality are important PISCES attributes, we have restricted or limited those operations and features that cannot be easily supported on all environments in an identical way. Adding support for another hardware platform or operating system and identifying the operations and functionality required is discussed later in this paper in the section, Extending PISCES to Additional Platforms and Environments.

## 3. The PISCES Source Code Engineering Methodology

The source code engineering methodology behind PISCES is nothing fundamentally new or revolutionary. Rather, it is a combination of common sense and ideas based upon the experience of many people in building real-world applications on several different types of hardware platforms. We identified the most important features and typical problem areas common in large software projects in order to try to put together a system to efficiently handle them. As engineers, we did not want to be hampered by a large and obtrusive mechanism that would be bypassed and ultimately wither except for the dictate of a manager. On the other hand, having seen many of the problems first hand, we realized that a solution would probably require the imposition of some structure and organization. Thus, the PISCES source code engineering methodology addresses the following points:

- Directory structure and organization
- Source code control and revision system
- Isolation of machine and operating system dependencies
- Unit and system level regression test code

---

SunOS is a trademark of Sun Microsystems, Inc.
Microsoft is a registered trademark of the Microsoft Corporation.
C/2 is a trademark of International Business Machines Corporation.

### 3.1. Directory Structure and Organization

A large software product can typically have hundreds of header and source files. As a result, nested subdirectories should be used to organize the files and simplify the understanding of related components through association in the same module. PISCES supports nested and recursive subdirectory make file invocation for all development activities though a single statement. A build activity can be activated at the top of the source tree, in which case all subdirectories will be affected. Alternately, a single subdirectory at some intermediate point in the source tree can be worked with independently. Finally, an operation can be performed on the files in the current directory, with subdirectories below subsequently processed for the same operation.

### 3.2. Source Code Control and Revision System

We strongly believe that all software systems should be developed and maintained within the structure of a source code control and revision system. PISCES uses RCS and automatically generates rules and dependency lists to support this system. As engineers, however, we strongly recognize the way in which such a system can hamper the productivity of a programmer who is forced to go through bureaucratic approval measures just to change a comment or make a minor change. As a result, PISCES makes use of the distinction between major and minor revision numbers supported by RCS. A major revision number precedes the decimal point; a minor revision number follows the decimal point. Thus, a file with revision 3.27 would indicate minor revision 27 after the third major revision level starting source base.

Major revision numbers should mirror significant functionality changes, completion of milestones, or releases to beta sites and customers. These can and should be strictly controlled by a project manager or leader. Minor revision numbers, on the other hand, should identify intermediate check points for source code modifications made by the engineering team between major revision levels. This offers the benefit of a frequent and traceable change history without the burdens and overhead of approval cycles and paperwork. When a source code base is "reved-up" to the next major revision number, all files in the system are automatically incremented to the next revision, whether or not there has been a change. PISCES generates rules in the make file to allow past major revision levels to be built, thus allowing a single source base to be used for both supporting older versions of software as well as maintaining the current version.

### 3.3. Isolation of Machine and Operating System Dependencies

The key motivation for PISCES is portability across heterogenous hardware and software platforms. As a result, we assume that a large portion of the source code for this system is machine independent. Thus, most source code files are located in the main module subdirectories. Nevertheless, the necessity for machine and operating system specific code is a common requirement for efficient, complex applications. However, the use of many #ifdef statements to handle such machine dependencies is strongly cautioned against. It is our belief that #ifdef statements should be used to identify and handle very minor differences across platforms, for example the inclusion of function prototypes for ANSI C compliant compilers or the identification of a missing function in a standard header file for a particular operating system. Significant differences, such as pathname construction/parsing or memory allocation schemes, should be isolated in a machine-specific subdirectory whose name is the same as that used in PISCES. The following platform names are currently defined:

| | |
|---|---|
| mips | -- DEC/MIPS workstation running Ultrix |
| sparc | -- Sun SPARC workstation running SunOS |
| vms | -- DEC VAX machine running VMS |
| sco386 | -- Intel 80386 platform running SCO Unix |
| os2 | -- PS/2 running IBM OS/2 extended edition |
| ti1500 | -- TI S1500 running TI System V |

Thus, a software subdirectory structure would have at or near the top level a machine subdirectory that contained an Imakefile and subdirectories whose name is the name of a particular PISCES-supported platform. When a software system is built, only that subdirectory whose name matches the system-type name defined in the top level Imakefile is built. Other platform subdirectories remain untouched. This isolation organization has two benefits. First, when porting a software source base to a new platform, most if not all machine-specific

functionality is already identified. Second, the remaining source code is relatively uncluttered with #ifdef statements, thus making for a more readable and understandable source code base.

### 3.4. Unit and System Level Regression Test Code

We believe the importance of unit-level and system regression test code cannot be stated strongly enough. Individual members of a project should spend a significant portion of their time writing unit-level test code for the areas and modules for which they are responsible. System level test code should be coordinated by a full time software quality engineer and every project member should participate in its design and development. The PISCES rules and actions support a system-level test code subdirectory and unit-level test code subdirectories below each module subdirectory. Specific rules to build the test code and run regression tests are directly supported and automatically inserted into a generated make file.

## 4. The PISCES Components

PISCES consists of the following five components: the Revision Control System (RCS), the macro-make file facility (imake), an automatic file dependency list generator (mkdepend), the DECUS ANSI C preprocessor, and the GNU diff program. All components are available in the public domain with no fees or royalities attached. In addition, all are written in C and of exceptional high quality. Although some of these components are typically available on an individual system, we decided to use the PISCES versions in all cases so as to assure the same interface, functionality, and features across all platforms. We believe that PISCES should be made available in the public domain for others who might find it useful. The following sections briefly discuss each component and identify modifications and changes made to implement PISCES on all platforms.

### 4.1. The Macro Make Facility (imake)

Imake is a tool available on many UNIX platforms that assists with the task of building a software system consisting of many files with a large number of dependencies. Imake was originally developed by Todd Brunhoff for the MIT X11 source distribution as a utility to simplify the process of configuring and building the X window system on various flavors of UNIX[1]. Imake uses the C preprocessor on a macro-makefile (the Imake file) to generate a make file for a particular system. Imake uses a predefined template file for default values and commands, a site file for system-specific pathnames and idiosyncracies, a project file for project-specific command names and procedures, and a set of support macros for tying everything together. An Imake file is system-independent; support for a new operating system or platform requires only the addition of a template file for that system. The Imake file specifying the dependencies and relationships between files in the software system to be built does not change. This allows machine dependencies (such has compiler options, alternate command names, and special make rules) to be kept separate from the descriptions of the various items to be built.

As available on the X11R4 source tape, Imake provides support for a number of UNIX-based platforms and environments to ease the building of X on a particular system[2]. However, Imake makes several assumptions about the platform on which it is being run, the most obvious of which are pathname construction/manipulation, the existence of a named pipes facility, and the availability and location of a C preprocessor. In addition, the collection of macros provided is both X- and UNIX-specific in nature and confusing to the novice user. As such, it is unsuitable for use on such platforms as OS/2 and VMS or in situations where the programmer is not an expert on the intricacies of a C preprocessor and the UNIX operating system. The PISCES version of Imake makes no such assumptions and has been rewritten to use "plain vanilla" C code that can be compiled and executed on most systems with a C compiler. In addition, the macros used by a programmer in an Imakefile have been simplified and restructured to provide support for simultaneous development of both C and C++ object files, parameterized pathname syntax and construction, and removal of operating system and X-specific features such as shared libraries and server/client operations.

Two other significant modification made to Imake for PISCES are the addition of several special characters used to represent certain behavior and the shortened file names of the various imake configuration files. Due to the behavior and idiosyncracies of the C preprocessor, Imake uses the character strings "@@\" to indicate a continuation line in an Imake rule, "@+" and "@-" to indicate that Imake should increment or decrement,

respectively, the immediately following number, "@!" to indicate that a string should be quoted as appropriate for a particular operating system, and "@#" to indicate that a line is a make file comment line. The names of the Imake configuration files have been modified to make them portable to non-UNIX platforms. The file names are listed below, with the modified name shown on the left and the original X11R4 file name given on the right:

| | |
|---|---|
| copyrite.imk | <no equivalent file> |
| *platform.cf* | <platform configuration file> |
| site.imk | site.def |
| project.imk | Project.tmpl |
| imake.imk | Imake.tmpl |
| imake.rul | Imake.rules |

The first file inserts a company-specific copyright notice into each generated make file. The second file is the platform-specific configuration file to override default PISCES commands and macros. The third file allows for specification of any site parameters, such as the location of the standard C and C++ header files. The fourth contains file names, pathnames, and other application-specific information. The fifth is the generic Imake template that controls the order and type of information used to create a make file. The last contains the default Imake rules and macros. When Imake is run, it searches for these files along the include path search directories, looking first in the top level project subdirectory and then in the main PISCES configuration directory.

### 4.2. The Automatic File Dependency Generator (mkdepend)

The make depend utility on the X11R4 source tape is associated with Imake and follows the include statements through each source file to determine all other files in the system that each is dependent upon. This dependency list is then appended to the generated make file so that if one or more dependent files are modified, the affected source file can be recompiled accordingly. As with Imake, however, there are a number of assumptions about the system and the nature of the dependencies, particularly concerning pathname construction, file name extensions, and the structure of the dependency list.

The PISCES version of mkdepend performs the same operation as the original version, but does so in a more platform independent manner. In addition to changes necessary to allow for compilation and execution on all hardware platforms and software environments, the most significant modification made is the manner in which pathnames are constructed and manipulated. In particular, several additional command line options provide for specification of starting and ending pathname strings, support for generation of RCS dependencies, and the pathname separator character. Finally, assumptions and changes to file name extensions are made to allow dependency lists to be created for RCS, header, C source, C++ source, and object files. Control and invocation of mkdepend is handled by Imake and is completely hidden from the programmer.

### 4.3. The DECUS ANSI C Preprocessor (cpp)

Many C and C++ language implementations separate the preprocessor and compiler functions. Others, (such as the Glockenspiel C++ language system or the IBM C/2 compiler), combine the preprocessor and compiler into one step. Since we needed a portable utility to massage macro make files that works under both scenarios, we decided to select an ANSI C-preprocessor that could be included as part of the PISCES tool set. Thus, the PISCES preprocessor is derived from and based upon a public domain C-preprocessor (the DECUS C preprocessor) made available by the DEC User's group and supplied on the X11R3 source tape from MIT. It has been modified to run on each of the target platforms and to comply with the draft ANSI C specification[3] with the exception that trigraph sequences are not supported.

### 4.4. The Revision Control System and File Differentiator (RCS and diff)

The Revision Control System (RCS) is a software tool written by Walter F. Tichy at the Department of Computer Sciences at Purdue University available on a wide variety of UNIX systems that assists with the task of keeping a software system consisting of many versions and configurations well organized[4]. Widely considered to be an efficient next-generation source control system to the popular SCCS, RCS manages revisions of text documents, in particular source programs, documentation, and test data in a space and time efficient manner. It

automates the storing, retrieval, logging and identification of revisions, and it provides selection mechanisms for composing configurations. For conserving space, RCS stores deltas, i.e., differences between successive revisions. The RCS file format is ASCII text, portable from one system to another. Thus, an RCS file containing versions of a file created and edited on a UNIX system can be used on a VMS system with no conversion.

Version control is the task of keeping software systems consisting of many versions and configurations well organized. RCS is a set of commands that assist with that task. RCS' primary function is to manage revision groups. A revision group is a set of text documents, called revisions, that evolved from each other. A new revision is created by manually editing an existing one. RCS organizes the revisions into an ancestral tree. The initial revision is the root of the tree, and the tree edges indicate from which revision a given one evolved. Besides managing individual revision groups, RCS provides flexible selection functions for composing configurations. RCS supports both major and minor revision levels. Typically, major revision numbers are for significant changes/updates or release versions. Minor revision numbers are used when making incremental changes that are limited in scope and nature.

RCS also offers facilities for merging divergent versions of a common file and for automatic identification. Identification is the 'stamping' of revisions and configurations with unique markers. These markers are akin to serial numbers, telling software maintainers unambiguously which configuration is before them. The merge capability allows two different but ancestorally related versions of a file to be merged so long as there are no competing changes and conflicts. This allows two engineers to each modify a different function in a source file implementing two functions. When each has finished modifying their respective function, the files can be merged to restore the single source file reflecting the changes made by each. The RCS merge capability requires a file differentiator tool for identifying the differences between two files. The GNU diff program is used as a portable and efficient tool for this purpose.

The PISCES versions of RCS and diff are functionally unaltered from their original UNIX implementations. Changes necessary to each were largely restrained to file names and extensions, examination of date/time stamps, and file reading and writing operations.

## 5. Installing and Bootstrapping PISCES

When first installed on a new system, the PISCES components must be bootstrapped by a system administrator or programmer. This typically requires access to system subdirectories and knowledge of system configuration, compilation, and installation procedures. The minimal PISCES components require no special compilation procedure. Each component is in its own subdirectory, with source and header files are under RCS control. A checked out version of each file is provided for bootstrapping purposes. The components should be built in the following order: cpp, imake, and mkdepend. Once complete, the minimal required components are available with which to build the trickier RCS and diff utilities. Finally, rebuild all PISCES components with themselves using the supplied Imake files to insure that everything is working correctly. Complete documentation and details for each platform can be found in the README file in the top level PISCES subdirectory.

## 6. PISCES Rules and Variables

A programmer controls the actions to be taken in an Imake file by using the PISCES rules defined as preprocessor macros in the Imake project, site, template, and configuration files. There are approximately 60 such macros defined, of which only a small number are actually used by the end-user programmer. Most are macros used internally for breaking down and structuring the actions supported by other higher-levels macros. An Imakefile contains calls to several of these higher-level macros to define the following items:

- optional top level and subdirectory specifications
- a required main or top level target/action
- a required list of header, source, and object files
- one or more required compilation rules
- one or more required program or library targets
- optional header and regression test operations

The most commonly used macros control basic operations such as selecting the type of object file to produce, specifying library, program, and directory names, and building programs. The best way to become familliar with these macros is to look at the examples in the tutorial below and study the results in the generated make file. The following list of macros identifies the main rules used by most engineers using PISCES:

| | |
|---|---|
| All() | - Simple target |
| All[1-24]() | - Simple target |
| LinkIncludes(files) | - Link headers to $(INCDIR) |
| NormalCObject() | - Normal C compile rule |
| NormalCPlusObject() | - Normal C++ compile rule |
| DebugCObject() | - Debug C compile rule |
| DebugCPlusObject() | - Debug C++ compile rule |
| TestCObject() | - Test code C compile rule |
| TestCPlusObject() | - Test code C++ compile rule |
| OptimizeCObject() | - Optimize C compile rule |
| OptimizeCPlusObject() | - Optimize C++ compile rule |
| LibraryName(name) | - Add/Update $(OBJS) to library |
| QuoteName(name) | - Add quotes around file name |
| VersionName() | - Generate RCS version name |
| Clobber() | - Remove object/scratch files |
| Clean() | - Delete objects |
| RCSCheckOut() | - Checkout files from RCS |
| Test() | - Compile and run regression tests |
| Test[1-24]() | - Compile and run regression tests |
| CLexScanner(scanner) | - User-level LEX rule (C compatible) |
| CYaccParser(parser) | - User-level YACC rule (C compatible |
| CPlusLexScanner(scanner) | - User-level LEX rule (C++ compatible) |
| CPlusYaccParser(parser) | - User-level YACC rule (C++ compatible |
| CProgram(program) | - Link objects into C program |
| CProgram[1-24](program) | - Link objects into C program |
| AuxillaryCProgram(program) | - Additional C program rule |
| CPlusProgram(program) | - Link objects into C++ program |
| CPlusProgram[1-24](program) | - Link objects into C++ program |
| AuxillaryCPlusProgram(program) | - Additional C++ program rule |
| TopLevelBootStrap() | - Bootstrap Imake from top level directory |

Each macro performs an operation or set of functions for a given target. When an operation is complete in the current directory level, the same operation is recursively performed on subdirectories if appropriate. The versions of macros with the number one through twenty four enclosed with brackets allow for the specification of up to 24 separate programs in a single Imake file. If there are three related C++ programs in a single Imake file, the programmer would use the **All3()** and **CPlusProgram3()** macros with the appropriate arguments. See example two below for more details. If you wish to add rules or procedures of your own for a specific project file to the collection of Imake rules, examine the macro definitions and the use of the special Imake command characters in the Imake rules file **imake.rul** located in the PISCES configuration subdirectory.

In addition to using PISCES rules, a programmer often needs to set or override system, project, or local values for preprocessor definitions, include search directory paths, and libraries. The following list of variables identifies the common variables most often needed by engineers using PISCES:

| | |
|---|---|
| STD_C_INCS | - Standard C include search directories |
| STD_C_DEFS | - Standard C command line symbol definitions |
| STD_C_LIBS | - Standard C archive libraries |
| STD_C_LIBDIRS | - Standard C library search directories |
| STD_CPLUS_INCS | - Standard C++ include search directories |
| STD_CPLUS_DEFS | - Standard C++ command line symbol definitions |
| STD_CPLUS_LIBS | - Standard C++ archive libraries |
| STD_CPLUS_LIBDIRS | - Standard C++ library search directories |
| PROJECT_C_INCS | - Project C include search directories |
| PROJECT_C_DEFS | - Project C command line symbol definitions |
| PROJECT_C_LIBS | - Project C archive libraries |
| PROJECT_C_LIBDIRS | - Project C library search directories |
| PROJECT_CPLUS_INCS | - Project C++ include search directories |
| PROJECT_CPLUS_DEFS | - Project C++ command line symbol definitions |
| PROJECT_CPLUS_LIBS | - Project C++ archive libraries |
| PROJECT_CPLUS_LIBDIRS | - Project C++ library search directories |
| LOCAL_C_INCS | - Local C include search directories |
| LOCAL_C_DEFS | - Local C command line symbol definitions |
| LOCAL_C_LIBS | - Local C archive libraries |
| LOCAL_C_LIBDIRS | - Local C library search directories |
| LOCAL_CPLUS_INCS | - Local C++ include search directories |
| LOCAL_CPLUS_DEFS | - Local C++ command line symbol definitions |
| LOCAL_CPLUS_LIBS | - Local C++ archive libraries |
| LOCAL_CPLUS_LIBDIRS | - Local C++ library search directories |

Each variable controls the value of a specific option at compile or link time. Note that there exists a version of each variable for both C and C++ for the standard, project, and local values. The PISCES macros and variables are more fully explained in the following section, A PISCES Tutorial.

## 7. A PISCES Tutorial

Most source code configuration problems follow one of a few patterns: compiling one or more source code files to be linked together to create a program; compiling one or more source code files to be archived in a library; creating one or more programs comprised of source code in one or more subdirectory modules; various maintenance activites such as cleaning up work files, running regression tests, or removing all object files and executables. This tutorial looks at several of these cases and discusses the flexibility and ordering of PISCES rules in an Imake file.

### 7.1. Example 1: A Simple C Program

Suppose you want to make the C program **prog** by compiling the three three source files **file1.c**, **file2.c**, and **file3.c** and linking the resulting object files together with the standard C runtime library. The following PISCES Imake file could be used:

```
HDRS =
SRCS = file1.c file2.c file3.c
OBJS = file1.o file2.o file3.o
All(prog)
OptimizeCObject()
CProgram(prog)
```

Lines one through three identify the header, source, and object files for the program. Notice that although there are no program-specific header files, we nevertheless include an empty definition. Line four identifies the main or top-level target. For historical reasons, this is given the label ‘all’ in the generated make file. Line five indicates that we want Imake to generate commands to create optimized C object files. Other possible choices are for standard and debug versions of C object files, in addition to the three analagous C++ object files. The last

line generates commands to check out source code from RCS, compile, link, clean, and install the program. All program names, command line options, and procedures for the various phases of this process are hidden in the generic template file, optional project and site files, and the machine-specific configuration file.

### 7.2. Example 2: Multiple C++ Programs

In this slightly more complicated example, there are two C++ programs each comprised of several source files. The second program has an associated header file and all source files are compiled with a command line definition for a symbol. The first program, **foo**, is made from the source files **file1.C**, **file2.C**, and **file3.C**. The second program is made from the source files **file3.C** and **file4.C**. The following PISCES Imake file controls their construction:

```
HDRS1 =
HDRS2 = foo.h
SRCS1 = file1.C file2.C file3.C
SRCS2 = file3.C file4.C
OBJS1 = file1.o file2.o file3.o
OBJS2 = file3.o file4.o
STD_CPLUS_DEFS = $(DFLAG)COMPANY=$(QUOTE)ABC Plumbing$(QUOTE)
All2(foo,bar)
OptimizeCPlusObject()
CPlusProgram1(foo)
CPlusProgram2(bar)
```

Lines one through six list the header, source, and object files for each program. There is one header file used by the second program and there are four C++ source files, one of which is used in both programs. Line seven defines a command line preprocessor symbol whose value is the character string "ABC Plumbing". Note that due to operating system differences, the command line "define" option and the mechanism for quoting the string are themselves macros. Line eight identifies the main targets, **foo** and **bar**, for the ‘all’ label. Line nine specifies that the program should be built from optimized C++ object files. Finally, lines ten and eleven generate the commands to checkout out source code from RCS, compile, link, clean, and install the two programs. As currently implemented, PISCES will allow a single Imake file to contain targets for upto 24 independent programs. This is accomplished by using the **HDRS**, **SRCS**, **OBJS**, **All**, **CProgram**, and **CPlusProgram** rules appended with the program number. In addition, library archives and/or program specifications can intermix C and C++ source and object files as necessary.

### 7.3. Example 3: A Library Archive With Yacc/Lex Dependents

In this last example, two C++ source files **file1.C** and **file2.C** are compiled with debug flags on and added to an application library. In addition, scanner and parser object files are also added to the library. The following PISCES Imake file controls this operation:

```
HDRS = program.h
SRCS = file1.C file2.C
OBJS = file1.o file2.o my_lex.o my_yacc.o
LEXSRC = lexer1.l lexer2.l
YACCSRC = decl.y tokens.y rules.y
All($(LIBRARY))
DebugCPlusObject()
OptimizeCObject()
LinkIncludes($(HDRS))
Library($(LIBRARY),$(OBJS))
CLexScanner(my_lex)
CYaccParser(my_yacc)
```

Lines one through three list the header, source, and object files. Lines four and five list the scanner and parser modules that, when concatenated together, processed, and compiled, produce the **my_lex.o** and **my_yacc.o**

object files. Line six specifies that the application library is the main make file target. The variable **LIBRARY** is specified in the project Imake file. Lines seven and eight indicate that the C compiler should generate optimized object files and the C++ compiler generate object files with the debug flag and symbols enabled. Line nine states that the header file(s) specified should be linked to the main application include subdirectory. Line ten controls the creation and update of the local library should any of the object files be changed. Finally, lines eleven and twelve control the generation of the scanner and parser modules with C-linkage using the public domain programs flex and byacc*.

### 7.4. Example 4: A Custom Imakefile For a Work Directory

So far, the discusion and examples in this paper have centered around project Imake files most often used in a batch mode or when rebuilding entire modules of a software system. Another common requirement is as a local Imake file in an engineers working subdirectory. Under this scenario, a programmer has several files that are undergoing modification. What is needed is a personalized Imake procedure that will allow for locally overriden values and alternatives to the default project and system variables. The following PISCES Imake file could control construction of several files in this manner:

```
HDRS = foo.h
SRCS = foo.C bar.C
OBJS = foo.o bar.o
RCSDEP =
LOCAL_INCS=$(IFLAG)$(HOME)
LOCAL_CPLUS_LIBS=$(LIBSFLAG)my_lib
LOCAL_CPLUS_LIBDIRS=$(LDIRFLAG)$(HOME)
All(hacker)
DebugCPlusObject()
CPlusProgram(hacker)
```

As in the previous examples, the first three lines establish the names of the header, source, and object files that comprise the program. Lines four through seven is the heart of this example. By default, PISCES assumes all source and header files are kept under RCS and automatically generates rules to examine such dependencies and check out files accordingly. For local working directories where a file is still under development, this is probably not convenient. By overriding the value of the **RCSDEP** symbol as in line four, PISCES does *not* look for RCS files to examine dependencies and check out the latest version. Lines five through seven set values for the local variables controlling include search directory, library, and library search directory. As a result, PISCES generates a Makefile that will check for include files first in the directory specified by **$(HOME)** before checking the project and standard include search directories. In addition, references for unresolved symbols at link time will be first searched for in the **my_lib** library archive before the project and system archives are searched.

By default, the values of the local and project variables are empty. A project leader edits the project file in the top level project directory to establish default values for the project variables. An individual engineer can provide his/her own values for these variables in a local Imake file by setting appropriate values as in the example above. In this manner, each engineer can configure Imake to work on Imake files while in a local development environment without affecting all other engineers on the project.

## 8. Using PISCES in the Software Development Process

After an Imake file is specified, the PISCES components can be used to create the machine-specific make file. Imake, the main program, requires four command line arguments: a preprocessor symbol identifying the machine type, another preprocessor symbol indicating the absolute pathname of the top level source directory for the software system being built, a revision number indicating the latest RCS major version number, and the include

---

* Flex is a public domain fast lexical analyzer generator compatible with lex(1). It was written by Vern Paxson of Lawrence Berkeley Laboratory and placed in the public domain. Byacc is a public domain yacc(1) compatible parser generator from the USC Berkeley Computer Science Department. Both programs are supplied in source format in the PISCES subdirectory and compile/execute on the PISCES-supported platforms.

file search directory path in which to look for the Imake configuration files. On a UNIX platform, a typical startup command might be:

```
% imake -Dsparc -DTOPDIR=/home/neath -DREV=1 -I/usr/local/bin/pisces
```

The machine name is used to select the appropriate command names and machine specific actions in the various configuration files. The RCS major revision number controls the number and type of rules generated for building previous versions of the software. Once an initial top level make file has been bootstrapped, the remaining system make files and dependency lists must be created by specifying the ‘bootstrap’ target to the make program. On a UNIX platform, this command would be:

```
% make bootstrap
```

Note that this is a special top level command available available only if the statement ‘#define TopLevelImakeFile’ is contained in the Imake file. After this step, the user need never use the Imake command again. All updates and modifications should be made to the Imake file, not the automatically generated make file. A new up-to-date make file can be created at anytime by specifying the ‘makefile’ target to the make program. On a UNIX platform, this command would be:

```
% make makefile
```

If the Imake file contains the statement ‘#define IHaveSubdirs’ indicating the presence of subdirectories, new make files for all subdirectories can be generated by specifying the ‘makefiles’ target. Other supported actions for a generated make file include all, checkout, install, clean, clobber, revup, test, and depend. Examine a PISCES-generated make file or refer to the example Imake files in the PISCES subdirectory for further details.

## 9. Extending PISCES to Additional Platforms and Environments

PISCES is intended to be a portable and dependable source code engineering system. As such, any system to which PISCES is ported should fully support all rules, actions, and behavior as on other existing platforms. This is necessary in order to insure that Imake files do not have to be altered from one platform to the next. To add support for another platform to those listed above, first compile and build the PISCES components. Next, add a machine-specific configuration file containing command names and/or equivalent functions as defined in the standard imake.imk template and imake.rul rules files. Since these files are included by Imake *after* this configuration file, platform-specific macros and command names can override the default PISCES values. Finally, modify the imake.imk template file to recognize the new machine type symbol and define the appropriate MacroFile symbol. Now rebuild the PISCES components using Imake to regenerate the make files and control the compilation process.

The difficulty of adding support for PISCES on additional operating systems and hardware platforms varies and cannot be predicted. Many UNIX-like operating systems should be fairly easy to support, but others may be quite difficult or even impossible. PISCES assumes that the system has a make facility with similar capabilities to the UNIX make(1) command. In addition, commands for compiling, copying and deleting files, and maintaining library archives are also necessary. Finally, memory availability may affect performance and even determine sucess or failure, particularly for operating systems with limited or non-virtual memory subsystems.

## 10. Conclusion

Texas Instruments is currently using PISCES on several large internal software projects that require multi-platform deliverables. We have found it to be a great aid in rapidly porting and maintaining a source code base from one platform to another. In addition, we have found that the use of PISCES reduces the need for a programmer to be an "expert" on the use of system-specific utilities on several platforms, not to mention the avoidance of peculiarities and idiosyncracies of particular utilities, ie. the nature of space characters versus tab characters in make(1).

## 11. Status of PISCES

PISCES is currently up and running on a Sun SPARCstation 1 (TM) running SunOS 4.x and a VAX 5400 MIPS machine running Ultrix release 3.1. The ports for a VAX 8600 running VMS version 5.1 and a PS/2 model 70/486 running OS/2 1.2 Extended Edition are currently in progress. Ports for the TI S1500 running TI System V and a PS/2 model 70/486 running SCO UNIX 2.3 will begin shortly. The SPARC and MIPS ports support the AT&T C++ translator (cfront) version 2.0 and the native C compiler, the SCO UNIX and OS/2 ports support the Glockenspiel translator version 2.0 with the IBM C/2 or Microsoft C compiler version 6.0, and the VAX VMS port supports the Oasys C++ translator version 2.0 and the native VAX C compiler.

## 12. References

[1] MIT X Consortium, *Imake Configuration Guide*, X11R4 Release Tape Documentation Notes

[2] Mark Moraes, *An Imake Tutorial*, Computer Systems Research Institute, University of Toronto.

[3] Brian Kernighan and Dennis Richie, *The C Programming Language*, Second Edition, Prentice-Hill, Englewood Cliffs, NJ, 1988.

[4] Walter F. Tichy, *RCS - A System for Version Control*, Department of Computer Sciences, Purdue University, West Lafayette, Indiana.

---

SPARCstation 1 is a trademark of Sun Microsystems, Inc.