WRPRC2 imake Configuration File Reference

Paul DuBois
dubois@primate.wisc.edu

Wisconsin Regional Primate Research Center
Revision date: 31 August 1993

This document describes in some detail the contents of the Release 2 WRPRC *imake* configuration files. A tutorial document is also available that describes how to write Imakefiles that use these files.

The first part of this document describes the contents of each of the configuration files. The second part discusses miscellaneous aspects of their use.

Imake.tmpl

Imake.tmpl is a template file that defines the architecture of the configuration files. It serves to `#include` the other configuration files in the correct order, as well as the *Imakefile* from the current directory, and it adds some default *Makefile* target entries. *Imake.tmpl* looks like this:

```
#include <Imake.p-vb>
#include <Imake.vb>

#define BeforeVendorCF
#include <site.p-def>
#include <site.def>
#undef BeforeVendorCF

#include ProjectVendorIncludeFile
#include VendorIncludeFile

#define AfterVendorCF
#include <site.p-def>
#include <site.def>
#undef AfterVendorCF

#include <Imake.p-params>
#include <Imake.params>

#include <Imake.p-rules>
#include <Imake.rules>

#include INCLUDE_IMAKEFILE
```

default *Makefile* target entries

`ProjectVendorIncludeFile` and `VendorIncludeFile` are defined in *Imake.vb* as the names of the vendor-specific configuration files appropriate for your system, e.g., *sun.p-cf* and *sun.cf*. `INCLUDE_IMAKEFILE` represents the *Imakefile*.

Configuration files are processed in pairs. The second member of each pair is a shared file located in the public configuration directory (e.g., */usr/lib/config/WRPRC2*), and is shared among all projects that use the WRPRC2 files. These shared files define the standard baseline configuration provided by the WRPRC2 files.

The first member of each pair is a project-specific file. When a project has configuration requirements such that it needs to override (or extend) the baseline information in the public file, it provides the necessary information in the project-specific file in its private *config* directory under the project root. To prevent file inclusion errors when a project doesn't provide particular project-specific files, the public directory also contains empty dummy copies of these files. That way, the project must provide only those files it needs, but those it provides take precedence over the dummy copies.

The following sections describe primarily the shared files because the default project-specific files are all empty. However, when it's likely you may want to override something in the shared files, that's pointed out. This means that you create a **.p-** file in the project's *config* directory and put the information there. You do *not* modify the dummy copy in the public configuration file directory.

Imake.vb

Imake.vb contains vendor blocks that determine the type of system you're using and define the macros `VendorIncludeFile` and `ProjectVendorIncludeFile` as the names of the vendor-specific configuration files appropriately. They also define symbols you can test to determine software and/or hardware type if necessary (e.g., in an *Imakefile*).

For instance, the Sun vendor block defines the vendor file names as *sun.cf* and *sun.p-cf*. It defines `SunArchitecture` (meaning the software is SunOS), and, depending on the hardware type, defines one of the symbols `SparcArchitecture`, `Sun3Architecture`, or `i386Architecture`.

If you port the WRPRC2 configuration files to a new platform, you must make sure there's an appropriate vendor block in *Imake.vb* and write a vendor file.

site.def

site.def contains site-specific definitions. This file is the primary candidate for modification when you install the WRPRC2 files, because you use it to express preferences for parameters such as your usual directory for installing user programs, etc. (You leave most of the other configuration files alone.)

The project-specific file *site.p-def* can be used when you want to override parameters on a site-specific basis for a particular project. Example: If `LocalBinDir` is set in *site.def* at your site to */usr/local/bin*, but you want to install programs built by a particular project into your own *bin* directory */u/you/bin*, put the following in the project's *site.p-def*:

```
#ifndef LocalBinDir
#define LocalBinDir /u/you/bin
#endif
```

Overriding Values in *site.def*

site.def and *site.p-def* are each split into two halves, like this:

```
#ifdef BeforeVendorCF
    /* stuff processed before vendor file */
#endif /* BeforeVendorCF

#ifdef AfterVendorCF
    /* stuff processed after vendor file */
#endif /* AfterVendorCF
```

You can specify override parameters in either half. A reason to put a parameter in the before half is that if you put something like the following in the after half:

```
#ifndef DirName
#define DirName /path
#endif
```

then the definition will be ineffective if the vendor file already defines `DirName`.

Alternatively, you can redefine `DirName` in the after half using a slightly different form, which is effective whether or not the vendor file defines it:

```
#ifdef DirName
#under DirName
#endif
#define DirName /path
```

However, this is inadvisable as it makes it impossible for individual projects to supply an alternative definition in *site.p-def*, defeating the purpose of the architecture's extensibility.

Thus, when you decide to override a parameter in *site.def*, take a look at your vendor file. If the vendor file overrides the parameter, that takes precedence over a definition in the site file. In that case, modify the vendor file instead.

vendor.cf

The files with names of the form **.cf* are vendor-specific files, e.g., *sun.cf*, *ultrix.cf*, *osf1.cf*. They're used to override the defaults for parameters specified in *Imake.params* when those defaults are incorrect.

A vendor file might also override the default definition of rules specified in *Imake.rules* if rules don't work properly on a given system. For example, *Mips.cf* overrides `InstallManFile()`, the rule that installs manual pages. Under Mips RISC/os, manual pages must be installed already *nroff*'ed, rather than in source form. So the rule is defined to run the manual page through *nroff* and install the result. An additional complication is that *nroff* does not by default come with RISC/os. Consequently, the rule is defined as a null operation if *nroff* isn't present on the system.

Project-specific **.p-cf* files are usually used when you define new project parameters in *Imake.p-params* but need to override defaults for a particular platform. This is analogous to the way **.cf* files override defaults specified in *Imake.params*.

Imake.params

Specifies default values for the general system and project parameters. Usually a parameter value is expressed as a *cpp* macro set to a default value that can be overridden in vendor or site files, and a *make* variable set to the value of the macro. For example:

```
#ifndef MvCmd
#define MvCmd mv
#endif
MV = MvCmd
```

Imake.params contains several kinds of information, described in the following sections.

General System Characteristics, Feature Symbols

This section describes the system type in broad terms and lists some particular features the system either does or doesn't have. The symbols `SystemV` and `SystemV4` indicate whether or not your system is based on System V Release 2 or 3, or System V Release 4. The default for both is `NO` (i.e., not based on System V at all). If that is incorrect for your system, override the defaults by defining the appropriate symbol as `YES` in the vendor file.

The `Has`*XXX* symbols indicate whether or not your system possesses certain capabilities or features. The defaults should be overridden in the vendor file as necessary.

Configuration File and Project Information

The symbols in this section describe the name, location, and release number of the WRPRC2 configuration files, the name and release number of the project, and the location of the project's private configuration files.

`ConfigRootDir` must be the same as the value used by *imboot* for the configuration file root directory. Override it in *site.def* if necessary.

The project name and release defaults are meaningless. Override them in the project's private *Imake.p-params* if you want them to be accurate.

General System Layout

This section defines things such as program, library, and administrative directories. One set of parameters describes the standard system layout — where such things as libraries, administrative files, user programs, and temporary files are found or created. However, many sites maintain a second set of directories into which locally installed files are put. For instance, you may put locally installed libraries and header files into */usr/local/lib* and */usr/local/include*. To accommodate this, another set of parameters describes the "local" system layout.

In general, when you write Imakefiles, you should use the local parameters because you can redefine them on a per-project basis in *Imake.p-params* or *site.p-def* without affecting where the standard system libraries are expected to be.

`BinDir`

The usual user program directory, default */usr/bin*.

`UsrLibDir`

Directory for libraries and library files. Default */usr/lib*.

`LintLibDir`

Directory for lint libraries. Default */usr/lib/lint*.

`AdmDir`

Directory for administrative programs and files. Default */usr/adm*.

`EtcDir`

Directory for servers and system maintenance tools. Default */usr/etc*; likely override value */etc*.

`IncludeRoot`

Top of system include file hierarchy. Default */usr/include*.

`TmpDir`

Directory for creation of temporary files. Default */tmp*; likely override values are */usr/tmp* or */var/tmp*.

`LocalRootDir`
`LocalBinDir`
`LocalUsrLibDir`
`LocalLintLibDir`
`LocalAdmDir`
`LocalEtcDir`

`LocalIncludeRoot`

Local directories corresponding to standard system directories. `LocalRootDir` is the root of the local directory hierarchy. The others are defined in terms of it, so you can move all of them by redefining it. To move just some of them, redefine individual `Local`*XXX*`Dir` parameters. If you want to make these directories equivalent to the corresponding standard system directories, define `LocalBinDir` as `$(BINDIR)`, `LocalUsrLibDir` as `$(USRLIBDIR)`, etc.

Manual Page Directory Layout

The parameters in this section define the layout of the manual page hierarchy on your system. In the simplest (default) case, this consists simply of the *man?* directories under */usr/man*. Some systems are more complex; see *Mips.cf* for an example.

`ManRoot` defines the root of the manual page hierarchy. `ManSourcePath` defines the name used to construct manual page section directories. The names of the individual directories are constructed by concatenating `ManSourcePath` with a suffix. The suffixes are given by the `Man?Suffix` parameters, and the directory names are given by the `Man?Dir` parameters, where *?* is 1, 2, 3, 4, 5, 6, 7, 8, *l* (el), or *n*. The parameters `ManSuffix` and `ManDir` indicate the "usual" suffix and directory (where user program manual pages are typically installed on your system.

There is no `LocalManRoot`, just `ManRoot`. This is because the manual page hierarchy already includes sections for local stuff (*manl* and *mann*). On the other hand, it's possible to argue that you'll want to install local stuff into an entirely separate manual page hierarchy, e.g., under */usr/local/man*. If so, override `ManRoot` in *site.def*.

Programs and Flags

This section defines parameters for programs like the C compiler, the loader, *rm*, etc., and flags that they use.

The complexity of `CFLAGS` is worthy of note; its structure is discussed a bit in the section *The Program-Building Model*.

Installation Support

This section defines parameters for installing various kinds of files. `InstallCmd` is the name of the program for installing things. By default it's *install*. That's suitable if your *install* is the BSD version, but if you have a System V *install*, `InstallCmd` is overridden as *bsdinst* or *install.sh* (shell scripts that emulate BSD behavior).

The other installation parameters define install flags and combinations of flags. The single flags are listed below. These are all *cpp* macros and all have corresponding uppercase *make* variables.

`InstStrip`

Flags for stripping executable binaries. Default is −*s*; override as the empty value if you don't want binaries stripped (which takes more space, but allows better debugging).

`InstOwner`
`InstGroup`

Owner and group names of installed files. The defaults are empty, since it's hard to choose good default ownerships that make sense across different systems. If you want files to be owned by a particular user or group, override the defaults. Note that you must include the −*o* or −*g* options with the name if you specify non-empty values:

```
#ifndef InstOwner
#define InstOwner -o name
#endif
#ifndef InstGroup
#define InstGroup -g name
#endif
```

Also note that ownership specifications may be ineffective if you're not *root* when you install files.

`InstProgMode`
`InstUidProgMode`
`InstGidProgMode`
`InstUGidProgMode`
`InstScriptMode`
`InstLibMode`
`InstDatMode`
`InstManMode`
`InstIncMode`

Mode flags for installing programs, setuid programs, setgid programs, setuid/setgid programs, executable scripts, libraries, data (non-executable) files, manual pages, and include (header) files. Installation of setuid and/or setgid programs may be ineffective unless you are *root* if you are not the installed owner and/or group.

The flag combination parameters (below) provide a convenience notation for concatenating common groups of flags. All of them reference `INSTOWNER` and `INSTGROUP`, and all the flags for executable binaries reference `INSTSTRIP`. In addition, each references a mode parameter for the appropriate type of file.

`InstProgFlags`
`InstUidProgFlags`
`InstGidProgFlags`
`InstUGidProgFlags`
`InstScriptFlags`
`InstLibFlags`
`InstDatFlags`
`InstManFlags`
`InstIncFlags`

Installation flags for installing programs, setuid programs, setgid programs, setuid/setgid programs, executable scripts, libraries, data (non-executable) files, manual pages, and include (header) files.

Document Preparation

This section defines parameters for programs used to format documents, such as *troff*, *tbl*, and *soelim*, and parameters for common *troff* macro packages.

WRPRC Libraries

At WRPRC, several locally-developed libraries are commonly used to build programs. The WRPRC2 configuration files provide link and dependency specifiers for them. The following table lists the specifiers provided by the WRPRC2 configuration files. For every library *XXX*, there is a *XXX*`Lib` variable for link purposes and a `Dep`*XXX*`Lib` variable for dependency purposes. There are *make* variables corresponding to each of these, which is what you refer to in Imakefiles.

Library Name	**Link Specifier**	**Dependency Specifier**

refer record manipulation library	`BibStuffLib`	`DepBibStuffLib`
Exception and termination manager library	`EtmLib`	`DepEtmLib`
Form processing library	`FplLib`	`DepFplLib`
Form query library	`FqlLib`	`DepFqlLib`
GECOS library	`GecosLib`	`DepGecosLib`
Simple log manager library	`LogMgrLib`	`DepLogMgrLib`
Simple memory manager library	`MemMgrLib`	`DepMemMgrLib`
Network database service library	`NdsLib`	`DepNdsLib`
Network I/O library	`NioLib`	`DepNioLib`
Order processing library	`OrderLib`	`DepOrderLib`
Sequence number library	`SeqNumLib`	`DepSeqNumLib`
Simple screen-management library	`SimScrLib`	`DepSimScrLib`
Token scanning library	`TsLib`	`DepTsLib`

By default, the link specifiers are of the form −*lname* and the dependency specifiers are empty. This reflects the assumption that all the libraries are installed in a system directory (e.g., `LOCALUSRLIBDIR`) on your machine. If you have a project that builds one of these libraries, you'll notice that it has an *Imake.p-params* file that overrides the specifiers for that particular library to point into the project tree. That's because the project can't assume the library is installed, and if it also builds applications that use the library, it must find the library within the project.

You'll also notice that the link specifier includes a reference to `LOADERLIBPREFIX`. Link libraries are typically specified in one of two ways, either by pathname or using −*l* notation (e.g., −*lmylib*). When a library is specified using a pathname, on most systems the pathname is sufficient. However, on Cray systems running UNICOS, it's necessary to use −*l* even for libraries specified by pathname. Thus, to write a reference to a library specified by pathname, you write it like this:

```
$(LOADERLIBPREFIX)$(TOP)/lib/libmylib.a
```

The variable `LOADERLIBPREFIX)` gets its value from the macro `LoaderLibPrefix`, which is normally empty. The default reflects that no prefix to the pathname is necessary in most cases. But *cray.cf* defines the macro as −*l* so the proper flag precedes the path.

This is a good example of how the exigencies of one kind of system can affect the way you write configuration files.

Imake.rules

The main function of *Imake.rules* is to define macros that generate *Makefile* target entries.

One motif that appears throughout the WRPRC2 rules is the implementation of rules in terms of other rules. One form of this is the construction of a rule as a concatentation of invocations of other rules. Another form is the implementation of groups of similar rules in terms of a single underlying general-purpose rule. An example of rule concatenation is `BuildProgram()`, which invokes `AllTarget()`, `RemoveTarget()`, `StuffToClean()`, and `LintSources()`. An example of use of a general-purpose rule is `InstallManFile()`, which provides a general interface to manual page installation. All the other man page installation rules are defined in terms of `InstallManFile()`.

The reasons this technique is used so much are: (i) to avoid writing a construct as the same literal text over and over when a rule is simpler to invoke; (ii) when a bug is found in a construct, it's easier to fix the rule that generates it than to fix multiple literal instances of the text; (iii) it's easier to change the behavior of rules that use the construct (e.g., by providing an alternate definition in a vendor file).

Non-Rule Macros

Imake.rules defines some macros which are not rules.

`NullArg`

This macro evaluates to nothing. Use it in rule invocations to specify an empty argument. (Don't just leave an argument empty, because some *cpp*'s complain or fail when an argument is expected but none is specified.)

`DirFailPrefix`

This macro is normally empty, but may be defined as − for those systems having a shell that doesn't evaluate the following constructs properly:

```
if [ -d ] ...
if [ ! -d ] ...
```

Normally `DirFailPrefix` is defined in vendor files as necessary, and is not of concern to the *Imakefile* writer.

Rule Macros

Rules are listed by sections, paralleling the layout by which *Imake.rules* is organized. Within sections, rules are described in the order they appear within *Imake.rules*, rather than alphabetically.

Directory-Making Rules

`MakeDir(`*`dir`*`)`

`MakeDir()` generates commands to check whether the given directory exists and to create it if not. This rule is used tab-indented as a command line; it's usually used to construct other rules, rather than in Imakefiles.

`MakeDirectories(`*`op,dirs`*`)`

This rule generates a target entry *`op`* that creates the named directories if they don't exist. Unlike `MakeDir()`, the rule may be used standalone, so it's suitable for use in Imakefiles.

Rules for Building Executable Programs

`BuildProgram(`*`prog,srcs,objs,linklibs,deplibs`*`)`

`BuildProgram()` is the general interface for building programs. It generates entries to build, remove, and lint a single program, and it may be invoked multiple times in the same *Imakefile*.

`prog` is the program name, *`srcs`* and *`objs`* list the program's source and object files, *`linklibs`* lists the libraries needed to link the program, and *`deplibs`* lists the libraries to check as dependencies. Dependency libraries must be given as pathnames.

`BuildProgram()` produces several targets. A *`prog`* target builds the program. An `all` target builds the program and any others in the *Imakefile*. A `clean` target removes the program and cleans up, `lint.`*`prog`* *lint*'s the sources for *`prog`*, and `lint` *lint*'s the sources for all programs in the *Imakefile*.

`BuildSimpleProgram(`*`name,linklibs,deplibs`*`)`

`BuildSimpleProgram()` is a simplified version of `BuildProgram()` that makes a number of assumptions, namely that you want to build a single program consisting of a single source file *name.c*, that the manual page, if there is one, is named *name.man*, and that you want to install the program in `LOCALBINDIR` and the manual page in `MANDIR`. `linklibs` and `deplibs` are as for `BuildProgram()`.

`BuildSimpleProgram()` sets the *make* variables `SRCS` and `OBJS`, invokes `BuildProgram()`, as well as installation and manpage installation rules, and invokes `DependTarget()`.

`BuildSimpleProgram()` may be invoked only once per *Imakefile*, and should not be used with any other program or library-building rules.

If you want to install the program in a different directory, you can assign a different value to `LOCALBINDIR` in the *Imakefile*:

```
LOCALBINDIR = /some/other/directory
BuildSimpleProgram(prog,NullArg,NullArg)
```

`SpecialCObjectCompile(`*`dst,src,flags,depends`*`)`

This rule allows you to specify special flags for compiling a particular C object file. *`dst`* is the object file name, *`src`* is the source file name, `flags` is the special flags to use when compiling it, and *`depends`* is any special dependencies for *`obj`* (in addition to *`src`*).

Library-Building Rules

The WRPRC2 rules support “normal” libraries only; there is no support for shared, debugging, or profiled libraries.

`NormalLibraryObjectRule()`

This rule redefines the *.c* → *.o* transformation *make* uses to compile C source files for building objects in libraries. It should be invoked exactly once in any *Imakefile* in which you invoke `BuildNormalLibrary()`. Since this rule may change the default transformation, it’s best not to build programs and libraries in the same directory.

`BuildNormalLibrary(`*`name,srcs,objs`*`)`

`BuildNormalLibrary()` generate a target entry to build a library from the named object files. When you use it, invoke `NormalLibraryObjectRule()` as well.

`name` is the library basename (e.g., for a library *libxxx.a*, the basename is *xxx*). *`srcs`* and *`objs`* list the library’s source and object files.

A `lib`*`name`*`.a` target is produced that builds the library. An `all` target builds all libraries in the *Imakefile*. A `clean` target cleans up. `lint.lib`*`name`*`.a` and `lint` targets lint the sources for the *`name`* library and all libraries in the *Imakefile*, respectively.

`BuildLintLibrary(`*`name,srcs`*`)`

Build a *lint* library for the given library based on the given source files. *`srcs`* should name the same file as the *`srcs`* argument to `BuildNormalLibrary()`.

Generates `llib-l`*`name`*`.ln` and `lintlib` targets that build the *`name`* *lint* library and *lint* library for all libraries in the *Imakefile*, respectively, and a `clean` target to clean up.

Rules for Building Targets by Running Templates through Filters

The WRPRC2 rules allow you to generate files from templates by running them through some filter program to process the template. The filter may be arbitrary, but for convenience, rules are provided that assume either *msub* or *cpp* as the filter.

The output file may be either executable (i.e., scripts) or non-executable. Executable scripts may be produced that, when run, are executed by an arbitrary script processor (not just the shell).

For all the rules, `dst` is the target file to produce, `src` is the input template, and `deps` names any dependencies that must exist before processing the template. For those rules that take a `filter` argument, it's the filter program that should process the template. For those rules that take a `prog` argument, it's the processor that executes the output script when it's run. `prog` must be a full pathname.

Rules that build executable scripts act differently according to whether `HasExecableScripts` is `YES` or `NO`. If it's `YES`, the rule writes a line like this at the beginning of the script, where `prog` is the script processor:

```
#!prog
```

If `HasExecableScripts` is `NO`, the rule writes a shell script that writes your script to a temporary file, passes the file to the script processor as a separate command, and removes the file. This means the script template contains no `#!` line at the beginning. It's added if necessary by the rule that generates the script. (This contrasts, for instance, with the X rules to generate scripts from *cpp* templates. For those rules, the `#!` line must be present in the template and is removed if necessary.)

`FileFromTemplate(dst,src,filter,deps)`

Produce a non-executable file from a template.

`ScriptFromTemplate(prog,dst,src,filter,deps)`

Produce an executable script from a template.

`FileFromMsubTemplate(dst,src,deps)`

Produce a non-executable file from a template, assuming *msub* as the filter program.

`ScriptFromMsubTemplate(prog,dst,src,deps)`

Produce an executable script from a template, assuming *msub* as the filter program.

`ShScriptFromMsubTemplate(dst,src,deps)`

Produce an executable script from a template, assuming *msub* as the filter program and the shell as the script processor.

`ScriptFromMsubTemplateWithFlags(prog,dst,src,flags,deps)`
`ShScriptFromMsubTemplateWithFlags(dst,src,flags,deps)`

Like `ScriptFromMsubTemplate()` and `ShScriptFromMsubTemplate()`, but these rules allow you to specify flags that are passed to *msub*. This is useful when you want to redefine the variable reference delimiters that *msub* recognizes. (See the *msub* manual page.)

`FileFromCppTemplate(dst,src,defs,deps)`

Produce a non-executable file from a template, assuming *cpp* as the filter program. `defs` contains any −*D*'s or −*U*'s you want to pass to *cpp*.

`ScriptFromCppTemplate(`*`prog,dst,src,defs,deps`*`)`

Produce an executable script from a template, assuming *cpp* as the filter program. *`defs`* is as for `FileFromCppTemplate()`.

`ShScriptFromCppTemplate(`*`dst,src,defs,deps`*`)`

Produce an executable script from a template, assuming *cpp* as the filter program and the shell as the script processor. *`defs`* is as for `FileFromCppTemplate()`.

Miscellaneous Target-Building Rules

`LinkTarget(`*`target,linkto`*`)`

Generates a target entry *`target`* that creates a link *`target`* to the file *`linkto`*.

`FileFromIntermediary(`*`dst,inter,src`*`)`

Generates *`inter`* from *`src`*, then updates *`dst`* if *`dst`* differs from *`inter`* or doesn't exist. Otherwise does not update *`dst`*.

`SimpleWorldTarget(`*`project,release`*`)`

Generates a simple `World` target entry so that you can say *make World* to build your entire project from scratch. Used only in the *Imakefile* in the project's root.

The `World` target is preceded by an empty `all` target so that `all` rather than `World` is the default target if no other rule precedes `SimpleWorldTarget()` in the *Imakefile* (which might be the case for a multiple-directory project).

Caveat: This rule uses a simple configure-clean-depend-build model of project building. If your project has any special requirements, you'll probably need to write your own `World` entry.

Installation Rules

Generally, the installation rules make use of the `INST*FLAGS` parameters, although there is no requirement for this. You can invoke `InstallFile()` and pass it any set of flags you want, e.g., for a setuid-*root* program, you can write this:

```
InstallFile(install,myprog,myprog,$(LOCALETCDIR),-o root $(INSTUIDMODE) $(INSTSTRIP))
```

See the section under *Imake.params* describing installation parameters.

For all these rules, the installation directory is created if it doesn't exist. That means it's prudent to run *make −n install* before you run *make install*.

`InstallFile(`*`target,dstfile,srcfile,dir,flags`*`)`

This rule is the general interface for installing files other than manual pages. Most other installation rules are trivial and simply invoke `InstallFile()` with arguments appropriate for different types of files.

`target` is the name of the target to produce, typically `install`. *`dstfile`* is the name of the installed file, *`srcfile`* is the name of the file to install, *`dir`* is the directory in which to install the file, and *`flags`* is the installation flags.

Besides the *`target`* target, an `i.`*`dstfile`* target is generated so that you can install individual files. Typically *`target`* is `install` so that *make install* installs everything.

Generally, *`dstfile`* and *`srcfile`* are identical. Two parameters are provided so you can install a file and rename it at the same time if you want. However, none of the rules in *Imake.rules* do so.

You can provide such a rule in *Imake.p-rules* if your project requires it.

`InstallProgram(`*`file,dir`*`)`

Install an executable (binary) program.

`InstallUidProgram(`*`file,dir`*`)`

Install a setuid executable (binary) program.

`InstallGidProgram(`*`file,dir`*`)`

Install a setgid executable (binary) program.

`InstallUGidProgram(`*`file,dir`*`)`

Install a setuid/setgid executable (binary) program.

`InstallScript(`*`file,dir`*`)`

Install an executable script. Differs from `InstallProgram()` in that the flag for stripping binaries isn't passed to *install*. Some versions of *install* complain if they're told to strip non-binary executables.

`InstallDataFile(`*`file,dir`*`)`

Install a non-executable file.

`InstallIncludeFile(`*`file,dir`*`)`

Install a C header file.

`InstallLibrary(`*`name,dir`*`)`

Install a library (as *lib`name`.a*), and run *ranlib* on the installed image if you have it on your system.

`InstallLintLibrary(`*`name,dir`*`)`

Install a *lint* library (as *llib-l`name`.a*). Generates targets `install.lint` and `i.llib-l`*`name`*`.a`. The former installs all *lint* libraries in the directory, the latter installs only *llib-l`name`.a*.

`InstallLink(`*`target,linkto,dir`*`)`

Make a link from *`dir/target`* to *`dir/linkto`*.

`InstallMultipleDataFiles(`*`target,files,dir`*`)`

Install a set of non-executables named by *`files`* into *`dir`*. *`target`* is the name of the target entry, usually `install`.

Rules for Installing Manual Pages

`InstallManFile(`*`target,name,dstsuffix,dir`*`)`

This is the general interface for manual page installation. `target` is the name of the target entry, usually `install.man`. In addition, an `i.`*`name`*`.man` entry is generated so you can install individual manual pages. *`name`* is the name of the manual page. The manual page file must end with a *.man* suffix, but you specify the name in the rule without the suffix. *`dstsuffix`* is the suffix to be given to the installed manual page. *`dir`* is the directory in which to install the manual page. It's created if it doesn't exist.

InstallMan1Page(*file*)
InstallMan2Page(*file*)
InstallMan3Page(*file*)
InstallMan4Page(*file*)
InstallMan5Page(*file*)
InstallMan6Page(*file*)
InstallMan7Page(*file*)
InstallMan8Page(*file*)
InstallManLPage(*file*)
InstallManNPage(*file*)

Install a manual page in the section 1, 2, 3, 4, 5, 6, 7, 8, *l* (el), or *n* directory.

InstallManPage(*file*)

Install a manual page in the ‘‘usual’’ manual page directory. By default, this is section *l* (el), but may be different at your site. It’s changed by changing the value of MANSUFFIX.

Pseudo-Installation Rules

These rules generate install target entries that simply announce the target isn’t installed anywhere. This has the function of making it explicit that the target is not intended to be installed, and that you didn’t just forget to invoke an installation rule in the *Imakefile*.

FakeInstallFile(*file*)

Generate a fake install target for a file. Also generates an i.*file* entry.

FakeInstallLibrary(*name*)

Generate a fake install target for a library. Also generates an i.lib*name*.a entry.

Rules That Generate Recursive Target Entries

The rules in this section generate target entries that run a given *make* operation in a set of subdirectories. They’re convenient for constructing directory-traversing operations. Executed in a project root, they will traverse the entire project tree. Executed in some subdirectory of the project, they will traverse the project subtree under that subdirectory.

Many of these rules are invoked automatically by the last section of *Imake.tmpl* and thus need not be invoked in individual Imakefiles.

NamedTargetSubdirs(*op,dirs,verb,flags*)

This rule is the general interface to recursive entry generation; it’s invoked by most of the other recursive rules to do the work. *op* is the *make* operation to generate a recursive entry for, *dirs* names the directories to process, *verb* is the word that *make* uses to announce what it’s doing as it processes the entry (e.g., ‘‘cleaning’’, ‘‘installing’’), and *flags* is any flags that should be passed to *make* in subdirectories.

Note: The WRPRC version of NamedTargetSubdirs() is like the X rule of the same name, but has one fewer argument.

MakeSubdirs(*dirs*)

Generate a recursive `all` target entry for building targets.

`DependSubdirs(`*`dirs`*`)`

Generate a recursive `depend` target entry for generating header-file dependencies.

`InstallSubdirs(`*`dirs`*`)`

Generate a recursive `install` target entry for installing programs and other non-manual-page files.

`InstallManSubdirs(`*`dirs`*`)`

Generate a recursive `install.man` target entry for installing manual pages.

`CleanSubdirs(`*`dirs`*`)`

Generate a recursive `clean` target entry for removing garbage files.

`TagSubdirs(`*`dirs`*`)`

Generate a recursive `tags` target entry for generating *TAGS* files.

Makefile Generating Rules

`MakefileTarget()`

Generates a `Makefile` target entry to rebuild the *Makefile* in the current directory. This rule invokes *imboot*, which is more portable than generating an *imake* command, since *imboot* takes care of knowing where the configuration file directories are on your machine. As such, *make Makefile* usually works to bootstrap a *Makefile* even when you move a project to a different machine, which would not be true were the entry to invoke *imake* directly.

`MakefileSubdirs(`*`dirs`*`)`

Generate a recursive `Makefiles` target entry for building Makefiles. The operation of this is described in a later section.

Document Formatting Rules

These rules generate target entries that process *troff* documents to produce PostScript files, which reflects my preferred output format. It would be easy enough to adapt the rules to produce plain text, RTF, etc., or to send the output directly to a printer.

`TROFFOPTS` may be set on the *make* command line to pass additional options to *troff*. For instance, if you want to format only pages 7 through 10 of a document, you can do this:

```
% make doc.ps "TROFFOPTS=-o7-10"
```

`TroffToPostScript(`*`dst,src,formatter,filters,deps`*`)`

This rule is the general interface to document processing. *`dst`* is the name of the output file, *`src`* is the name of the input file or files, *`formatter`* is the formatting program and its options (usually some *troff* program), *`filters`* names any preprocessors, and *`deps`* names any dependencies that must exist before processing the document.

`filters` can either be `NullArg` if there are no preprocessors, or else a piece of a command pipeline such as `|$(TBL)` or `|$(TBL)|$(EQN)`. Note the use of the `|` character; if non-empty, *`filters`* must begin with one.

This rule generates a *`dst`* target to produce the formatted output file, and a `clean` target to remove it.

`TroffMsToPostScript(`*`dst,src,filters,deps`*`)`
`TroffMeToPostScript(`*`dst,src,filters,deps`*`)`
`TroffMmToPostScript(`*`dst,src,filters,deps`*`)`
`TroffManToPostScript(`*`dst,src,filters,deps`*`)`

These rules simply invoke `TroffToPostScript()`, passing their arguments to it and adding the appropriate macro package argument (−*ms*, −*me*, −*mm*, or −*man*).

Odds and Ends

`RemoveTarget(`*`target`*`)`

This rule generates commands to remove the target (a file). This rule is used tab-indented as a command line; it's usually used to construct other rules, rather than in Imakefiles.

`RanLibrary(`*`args`*`)`

This rule generates commands to run *ranlib* if it exists on your system, or an empty command otherwise. *`args`* lists any flags to be passed to *ranlib*. This rule is used tab-indented as a command line; it's usually used to construct other rules, rather than in Imakefiles.

`UpdateIfDifferent(`*`dst,src`*`)`

Updates *`dst`* from *`src`* if *`dst`* doesn't exist or is different than *`src`*. Otherwise does nothing. This rule is used tab-indented as a command line; it's usually used to construct other rules, rather than in Imakefiles.

`StuffToClean(`*`stuff`*`)`

This rule generates a `clean` entry with *`stuff`* as the dependency (multiple files may be listed). It is used in rules that build targets that should be removed when *make clean* is executed. You can invoke it explicitly in your *Imakefile* to make sure a given file is removed when the `clean` operation is performed.

`AllTarget(`*`target`*`)`

This rule generates an `all` entry with *`target`* as the dependency. It is used in rules that build targets which should be built when *make all* is executed. It may be used in an *Imakefile* to indicate additional targets you want to make sure get built when the `all` operation is performed.

`LintSources(`*`prog,srcs`*`)`

Generates a `lint.`*`prog`* target entry to run the sources for *`prog`* through *lint*. *`srcs`* should name the program's source files. Also generates a `lint` target entry; *make lint* runs *lint* for all programs in the *Imakefile*.

`DependTarget()`

Generates a `depend` target entry to generate header-file dependencies. *make depend* should be run after building or rebuilding the *Makefile*.

This rule assumes the *make* variable `SRCS` is set to the names of the source files to be processed.

`CleanTarget()`

Generates a `clean` target entry to remove garbage files. This is invoked by *Imake.tmpl*, so you don't need to. The files that are removed are defined by `FilesToClean`.

`TagsTarget()`

Generates a `tags` target entry to create a *TAGS* file.

How the WRPRC2 `MakefileSubdirs()` Rule Works

The `MakefileSubdirs()` rule is the trickiest rule in *Imake.rules*. Briefly, it works like this: for every directory it operates on, it changes from the current directory (the parent) into the subdirectory (the child), generates a *Makefile* there, runs *make Makefiles* with the new *Makefile* in case the child has subdirectories of its own, then returns to the parent directory.

Changing directories is relatively easy. A *cd* changes into the child, and the return to the parent is achieved by enclosing within parentheses the *cd* and the commands that run in the child, so that the commands run in a subshell. This way the *cd* applies only in the subshell and the current directory again becomes the parent when the subshell terminates.

The *Makefile* in the child is generated by running *imboot* and passing to it the values of the project root directory and the current directory within the project that are correct for the child. These are derived from the values of `TOP` and `CURRENT_DIR` in the parent directory. If `TOP` is an absolute path, then the value in the child is the same as the value in the parent. However, if `TOP` is a relative path from the parent to the project root, its value must be adjusted by prepending the path from the child to the parent.

The adjustment value is "`../`", "`../../`", "`../../../`", or "`../../../../`", depending on how many levels below the parent the child is located. The adjustment is determined from how many "`/`" characters are in the subdirectory's name. One complication is that if the name begins with "`./`", then that "`/`" doesn't count.

The location of the child within the project is easy to derive. `CURRENT_DIR` is the location of the parent within the project, so the location of any child *xyz* is `$(CURRENT_DIR)`*/xyz*.

Note that child directory names passed to `MakefileSubdirs()` must be specified as relative paths from the parent or this rule will not work. (But it would be absurd to specify them as absolute paths, anyway.)

The WRPRC2 `MakefileSubdirs()` rule is simpler than its counterpart in X11 Releases 1−5. Of course, by using *imboot*, the rule incurs the cost of invoking an additional shell script. However, some difficulties are avoided. For example, some alternative approaches are to invoke *imake* directly, or to run *make* recursively to execute a subsidiary target that itself generates an *imake* command. The success of either approach hinges on getting the values of the macros `TOPDIR` and `CURDIR` set properly in the child so that `TOP` and `CURRENT_DIR` are correct. Conceptually this is no different than figuring out the proper values of the project root and the current directory to pass to *imboot* in the child, but in practice is more difficult because it involves heavy interaction between use of *make* variables to set *cpp* macros and vice-versa. Using *imboot* involves none of this. An additional complication is that if the *imake* command specifies a configuration directory located within the project (i.e., specified in terms of `TOP`), the *−I* argument giving that location must usually be adjusted, too. This is ugly because the adjustment value must be inserted into the middle of the argument.

An additional simplification gained by using *imboot* involves the handling of existing Makefiles. Typically, *Makefile*-generating operations remove any existing *Makefile.bak*, then rename any existing *Makefile* to *Makefile.bak* before generating a new *Makefile*. If the *Makefile* target entry generates an *imake* command itself, it must also take care of this removing and renaming. This is unnecessary when *imboot* is used instead, since *imboot* handles *Makefile* and *Makefile.bak* itself. (The X11R5 rule `MakeNsubdirMakefiles()` illustrates clearly why you want the removing/renaming stuff in *imboot* and not in your *Makefile*.)

The Program-Building Model

A program is built by compiling its source files, then linking the resulting object files together (along with any libraries that may be needed) to produce the final executable. This section explains the model used by the WRPRC2 files to implement this process, with particular emphasis on compilation and loader parameters.

Compilation Parameters

The model for compiling C object files looks like this:

```
$(CC) -c compiler-flags file.c
```

The content of `compiler-flags` is given by the value of the *make* variable `CFLAGS`, which indicates several types of information. You don't set `CFLAGS` directly; instead you set the parameters that `CFLAGS` is defined in terms of. Two types of parameters of general interest are −*I*'s to specify where to look for include files, and −*D*'s and −*U*'s to define or undefine macros. The configuration parameters for these are shown in the table below. Mixed-case parameter names are *cpp* macros and uppercase names are *make* variables.

Type of Information	Global Parameters	Per-project Parameters	Per-Directory Parameters
Include files	`StandardIncludes` `STD_INCLUDES`	`ProjectIncludes` `PROJECT_INCLUDES`	`INCLUDES`
Defines	`StandardDefines` `STD_DEFINES`	`ProjectDefines` `PROJECT_DEFINES`	`DEFINES`

For instance, to specify include file directories, values may be assigned to the *make* variables `STD_INCLUDES`, `PROJECT_INCLUDES`, and `INCLUDES`. The three parameters are related, but have different functions:

- `STD_INCLUDES` is intended as a global parameter. It's the most far-reaching and applies to all projects configured with the WRPRC2 files. `STD_INCLUDES` gets its value from `StandardIncludes`, which should be given a default value that's appropriate for your system when you install the configuration files. You might, for instance, keep locally installed header files under `LOCALINCLUDEROOT` rather than under the standard include hierarchy root. This is information that's likely to be needed by many or most projects compiled on your system. To make sure that −*I*`$(LOCALINCLUDEROOT)` is passed to all *cc* commands that compile source files, `StandardIncludes` can be written like this:

  ```
  #ifndef StandardIncludes
  #define StandardIncludes -I$(LOCALINCLUDEROOT)
  #endif
  ```

  That is in fact the default value given in *Imake.params* in the WRPRC2 distribution. If you need to pass other directories in addition to `LOCALINCLUDEROOT`, override the default value of `StandardIncludes` in *site.def*:

  ```
  #ifndef StandardIncludes
  #define StandardIncludes -I$(LOCALINCLUDEROOT) -I/var/include
  #endif
  ```

  Or, if you install local include files in the usual system include hierarchy (not recommended), you can redefine `StandardIncludes` to be empty:

  ```
  #ifndef StandardIncludes
  #define StandardIncludes /**/
  ```

```
#endif
```

- `PROJECT_INCLUDES` is intended to specify include directories particular to a given project. It applies in every directory of your project. Its value comes from `ProjectIncludes`, which is empty by default. If necessary, override the default in *Imake.p-params* in your project's *config* directory.

  For instance, if your project has project-specific header files in *include* under the project root, you might write something like this to make sure the directory is passed to all compiler commands for the project:

  ```
  #ifndef ProjectIncludes
  #define ProjectIncludes -I$(TOP)/include
  #endif
  ```

- `INCLUDES` is intended to specify include directories that are particular to an individual directory. You assign it a value directly in the directory's *Imakefile*:

  ```
  INCLUDES = -Idir
  ```

For specifying *−D*'s and *−U*'s, the three parameters `STD_DEFINES`, `PROJECT_DEFINES`, and `DEFINES` are used. They're given values in a manner very similar to the way you specify include directory parameters. The defaults for the project- and *Imakefile*-specific parameters are empty. `StandardDefines` is `-DSYSV` or `-DSVR4` if your system is SVR2/SRV3 or SVR4, respectively, or empty if it's not based on System V. If `StandardDefines` needs to be non-empty, it's generally defined in your vendor file when you install the configuration files, rather than in *site.def.*. That's because defines needed on a system-wide basis tend to reflect peculiarities of your vendor's compiler rather than characteristics of your site's file system layout.

Include directory and defines parameters become part of the value of `CFLAGS`. Since the *Imakefile* is most specific, parameters specified in it should have the highest priority. Conversely, the global parameters are least specific, so parameters specified there should have the lowest priority. Project-specific parameters are in between.

For include directory parameters, earlier *−I*'s on a *cc* command take precedence over later ones, so include file parameters are ordered like this:

```
$(INCLUDES) $(PROJECT_INCLUDES) $(STD_INCLUDES)
```

On the other hand, later *−D*'s and *−U*'s take precedence over earlier ones, so defines are ordered like this:

```
$(STD_DEFINES) $(PROJECT_DEFINES) $(DEFINES)
```

Loader Parameters

The model for linking executable programs looks like this:

```
$(CC) -o prog objects loader-flags link-libraries extra-libraries
```

`prog`, `objects`, and `link-libraries` generally come from the invocation of the rule that builds the program (e.g., see the `BuildProgram()` rule). `loader-flags` and `extra-libraries` come from the configuration files.

- `loader-flags` specifies any special flags the loader needs in order to produce a final executable. For example, the linker looks for libraries in certain system directories by default. If you keep local libraries somewhere other than those directories, you need to specify search directories for the linker using *−L* arguments.

- `extra-libraries` is used to compensate for compatibility problems or library deficiencies on your system. For example, a non-BSD version of UNIX may allow programs that use BSD functions to be built by providing a BSD compatibility library (accessed, e.g., as *−lbsd*). This means on some systems you must add *−lbsd* to the link command.

`extra-libraries` is not for libraries you're certain you'll need. For instance, if a program *myprog* uses ETM functions, you know you'll need to link in the ETM library, and you might build the program by invoking `BuildSimpleProgram()` like this:

```
BuildSimpleProgram(myprog,$(ETMLIB),$(DEPETMLIB))
```

Like includes and defines, loader flags and extra libraries may be specified at three levels, as summarized in the following table.

Type of Information	Global Parameters	Per-project Parameters	Per-Directory Parameters
Loader flags	`StandardLoadFlags` `STD_LDFLAGS`	`ProjectLoadFlags` `PROJECT_LDFLAGS`	`LDFLAGS`
Extra libraries	`StandardLoadLibs` `STD_LDLIBS`	`ProjectLoadLibs` `PROJECT_LDLIBS`	`LDLIBS`

Loader flags are given in the same order as include directories (i.e., *Imakefile*-specific first, global last):

```
$(LDFLAGS) $(PROJECT_LDFLAGS) $(STD_LDFLAGS)
```

Extra libraries are specified after other libraries, *Imakefile*-specific libraries last:

```
$(STD_LDLIBS) $(PROJECT_LDLIBS) $(LDLIBS)
```

Tertiarities in the WRPRC2 Architecture

The WRPRC2 configuration file architecture (that is, the order in which configuration files are processed, and the locations in which to look for them) is designed explicitly to allow for a two-level specification of information: a standard baseline configuration specified by the shared files in the public directory, and project-specific configuration information specified in private files in the project's *config* directory.

However, the discussion in the previous section illustrates that in some instances there is allowance for a tertiary (three-level) configuration specification, e.g., in the way you specify include-file directories and preprocessor defines. The third level in these cases is per-*Imakefile*, that is, you can specify directories or defines that apply within a given *Imakefile* but nowhere else.