

# **VR Juggler**

## **The Build System**

**Patrick Hartling**

---

# VR Juggler: The Build System

Patrick Hartling

2.0.0

Published \$Date: 2007-01-01 10:26:55 -0600 (Mon, 01 Jan 2007) \$

Copyright © 2002–2007 Iowa State University

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being Appendix C, *GNU Free Documentation License*, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in Appendix C, *GNU Free Documentation License*.

---

---

---

---

# Table of Contents

Software Choreography .....	vi
I. Introduction .....	1
1. The Philosophy .....	2
2. Goals of Doozer++ .....	3
Ease of Use .....	3
Platform Independence .....	4
Compiler Freedom .....	4
Language Freedom .....	4
Separation of Tools and Tasks .....	4
Configuration .....	5
Compilation .....	5
Further Developments .....	6
3. Goals of the VR Juggler Build System .....	7
Centralize Complexity .....	7
Minimize Special Steps .....	7
Port Quickly to New Platforms .....	8
II. Design .....	9
4. The “Global” Build .....	10
Follow the Rules .....	10
Required Targets .....	10
File and Directory Names .....	13
Installation Hierarchy .....	14
configure.pl .....	14
Build Configuration File .....	15
Dependency Management .....	16
The *-config Scripts and the *.m4 Macro Files .....	16
A Typical -config Script .....	16
A Typical .m4 File .....	18
The Developer Installation .....	20
Future Goals .....	21
5. The Documentation Build .....	22
The Rules .....	22
Required Makefile Targets .....	22
Installation Hierarchy .....	22
Configuration and Customization .....	22
III. Extension .....	25
6. Extending the Global Configuration Script .....	26
JugglerConfigure.pm .....	26
JugglerConfigure .....	26
JugglerModule .....	26
ModuleDependency .....	27
configure.pl .....	27
Command-Line Argument Handling .....	27
Help Output .....	28
Build Configuration .....	28
File Regeneration .....	30
7. Extending a Module's Configure Script .....	31
Basic Concepts .....	31
Comments .....	31
Language .....	31
Variable Naming .....	32
Parameters to Macros .....	32
Custom Preprocessor Information .....	33

Structure of a <code>configure.in</code> File .....	33
Initialization .....	33
Custom Arguments .....	33
System-Dependent Setup .....	34
External Program Checks .....	34
External Library Checks .....	34
Header File Checks .....	35
Library Function Checks .....	35
Makefile Substitution .....	35
File Generation .....	35
Specific Modules .....	36
Tweak .....	36
8. Extending a Module's Makefiles .....	37
Makefile Conventions .....	37
File Names .....	37
Variable Assignments .....	37
Variable References .....	38
The Glue Makefile .....	38
Basic Structure .....	39
Auxiliary Files .....	45
Extension .....	46
Individual Component Makefiles .....	47
Basic Structure .....	47
Extension .....	49
Specific Modules .....	49
Multiple Libraries in VR Juggler .....	50
Java in Tweak .....	50
IDL in Tweak .....	50
IV. Appendices .....	51
A. Helper Scripts .....	53
<b>mtree(1)</b> and <b>mtree.pl</b> .....	53
<b>install-dir.pl</b> .....	53
<b>install-src.pl</b> .....	53
<b>makefiles-gen.pl</b> .....	54
InstallOps Perl Module .....	54
<b>make-ver.sh</b> .....	55
<b>mkmakefile.pl</b> .....	56
B. Build System Usage .....	57
C. GNU Free Documentation License .....	58
PREAMBLE .....	58
APPLICABILITY AND DEFINITIONS .....	58
VERBATIM COPYING .....	59
COPYING IN QUANTITY .....	59
MODIFICATIONS .....	60
COMBINING DOCUMENTS .....	61
COLLECTIONS OF DOCUMENTS .....	62
AGGREGATION WITH INDEPENDENT WORKS .....	62
TRANSLATION .....	62
TERMINATION .....	62
FUTURE REVISIONS OF THIS LICENSE .....	63
ADDENDUM: How to use this License for your documents .....	63
Glossary .....	64
Index .....	66

---

# Software Choreography

Within this book, we present the fundamentals of the VR Juggler 2.0 build system, a complex piece of software in its own right that manages the compilation of the various modules that make up VR Juggler 2.0 (and beyond, I suspect). Those few readers who may be familiar with the VR Juggler 1.0 build system may recognize a few similarities, but truly, the Doozer++-based build is quite different than what was originally intended to be backwards compatible with an IRIX-only build written for the first version of VR Juggler (then called VRLib).

Since 1998, the Autoconf-based VR Juggler build system has grown more and more complex. Originally, the build made use of Autoconf, GNU make, and a few small Perl scripts to simplify installations. Now, in mid-2002, the build system still makes use of all those tools, but it has many, many custom m4 macros (most of which come from Doozer++), makefile stubs, Autoconf-like scripts written in Perl, and custom CVS utilities. It may yet evolve to include Java- and Python-based tools.

With all this (growing) complexity, the VR Juggler 2.0 build system is a colossal effort in software choreography wherein all the pieces have to come together in such a way as to exude a Broadway-caliber performance. In this book, I strive to lay out the steps like one of Arthur Miller's finest teachers.

---

# Part I. Introduction

In sooth, I know not why I am so sad:

It wearies me; you say it wearies you;

But how I caught it, found it, or came by it,

What stuff 'tis made of, whereof it is born,

I am to learn;

And such a want-wit sadness makes of me,

That I have much ado to know myself.

--William Shakespeare, *The Merchant of Venice*, Act 1, Scene 1

In this first part, we explain the basics of the VR Juggler build system. We will begin with the goals of the Doozer++ software package which forms the basis for the entire build. We move on to the specific goals of the VR Juggler build system. The scope of the VR Juggler build, while broad, is narrower than that of Doozer++, and hence, it will be useful to understand first why Doozer++ exists and what purpose it serves.

---

---

# Chapter 1. The Philosophy

Nearly everything has a philosophy, and the VR Juggler build system is no different. The build system philosophy has evolved from the following observation: programmers think more about writing their code than about compiling it. Compiling takes time that could be spent writing code, so compiling must be fast. Command-line arguments have little to do with code, and thus they are easily forgotten. Paths to external dependencies are esoteric file system things that are never referenced in code. Because of these points, the basic philosophy of the VR Juggler build system and of Doozer++ in general is simple: automate everything.

Automation in a build system means doing as much as possible to avoid requiring programmers to type anything beyond the following familiar command:

```
configure ; make ; make install
```

Whether this is always possible is highly debatable, but either way, it should always be the goal. If the above cannot be achieved, some mechanism must exist so that programmers feel as though that is all they ever have to type.

There is much more to the philosophy of the VR Juggler build system than just automation, but someone looking at the code would see that steps to automate configuration and compilation make up a sizable percentage of the total line count. Beyond simplifying the process of building, we also have the goal of simplifying extensions to the build system.

Unfortunately, repeated experience has shown that no amount of simplification is enough to satisfy even the most patient of users. This harkens back to the original point stating that programmers focus more on writing their code than on compiling it. A build system is always going to be foreign to the majority of the people on any given team, and taking the time to learn and understand a build system takes up valuable coding time. Despite these bleak statements, it is possible to put together a build system that requires little effort to extend when a new file must be compiled or a new source directory is added. Those who hope for more than this are likely to be let down.

More details on the goals of Doozer++ and the VR Juggler build system are provided in the next chapter. All of these goals are founded by the philosophy discussed above.

---

# Chapter 2. Goals of Doozer++

When the first *Autoconf*-based VR Juggler build system was started in mid-1998, we had to make a decision: should we use *Automake* in addition to *Autoconf*? At the time, we felt that there were several problems with *Automake*, including the following:

1. The generated makefiles were too complex to debug
2. Use of compilers other than GCC was too difficult
3. Restrictions on the structure and contents of the source tree were undesirable

In four years, we have only been proven wrong on point #3. In our experience, *Automake* still generates extremely complex makefiles (though it does it very nicely from extremely simple input), and use of compilers such as the MIPSpro Compilers or Microsoft Visual C++ is still very hard.

Based on our needs for VR Juggler (and many other projects that have been developed at Iowa State University's Virtual Reality Applications Center), we identified the following as key goals for Doozer++:

- Simplify the use of *Autoconf*
- Allow the use of any operating system
- Allow the use of any compiler
- Allow code to be written in any language
- Use the best tool for a given task

Each of these goals will be addressed in the following sections.

## Ease of Use

Time and time again, we have seen a resistance to the use of *Autoconf* because of its (seemingly) arcane language constructs. While macro languages are difficult to use and **sh**(1) syntax is not immediately intuitive, neither tool can be considered arcane. Indeed, **sh**(1) is a much more powerful language than most other shell scripting languages, but because of its unique syntax, people tend to shy away from its use.

Further complicating the issue is the complexity of *Autoconf* in general. It contains many, many macros, and it defines rules for the order of executing those macros. In order to use *Autoconf* effectively, build system developers must read a fair amount of documentation (all of which is readily available). In our experience, however, many people feel that development build system software should be immediately obvious<sup>1</sup> and should not require much effort to learn.

As a result of these issues, we have attempted to make Doozer++ easier to use than “raw” *Autoconf*-based *<>configure scripts*. Doozer++ macros hide some common details that often trip up *configure* script authors. Many utility macros are provided to reduce code duplication between *configure.in* files. For example, code for verifying that an existing installation of a software package meets a version

---

<sup>1</sup>Of course, little, if anything, is “immediately obvious” in reality. Users tend to want something that is similar to existing tools or something that allows them to make use of existing knowledge.

requirement is provided. The code for performing the check is modularized to separate the common tasks involved with this. Namely, Doozer++ provides macros comparing two version numbers, acting on the results of the comparison, and caching the results to speed up future tasks. Each macro is used as the basis for the next so that users have the freedom to choose how they want their code to behave. We feel that this offers a level of flexibility not available with Autoconf 2.13.

## Platform Independence

Because VR Juggler has always been a cross-platform tool, we have had the need for a cross-platform way to build it. The VR Juggler 1.0 build system achieved this to a limited degree, and Doozer++ goes a step further toward true platform independence. At this time, Doozer++ makes use of software utilities found on all modern operating systems. The list of utilities includes `m4(1)`, `sh(1)`, Perl, Autoconf, and GNU make. We have avoided the use of platform-specific tools because it is all too easy for the platform-specific parts of a build system to get out of sync with each other. Furthermore, we have tried to avoid the use of platform-specific code whenever possible. This is a more difficult goal to achieve, but later sections will address the extent to which platform-specific code can be reduced.

## Compiler Freedom

As a cross-platform C++ library, VR Juggler must be compatible with the prevailing C++ compiler on a given platform. In the case of IRIX, that is SGI's MIPSpro Compilers. Similarly, the use of Microsoft Visual C++ is crucial on Win32 platforms. Open source operating systems such as FreeBSD and Linux use GCC, so we have not ignored that compiler whatsoever. Due to our limited needs, however, we have focused on compilers for the C, C++, Java, and IDL languages. Nothing prevents Doozer++ from being extended to allow compiling of code written in FORTRAN, Ada, Pascal, etc.

In Doozer++, an `m4` macro sets up the basic platform-specific pieces. User-level code (in an Autoconf `configure.in` file) then uses that information to execute other macros that pick the appropriate compiler. This builds upon the foundation provided by Autoconf for detecting installed programs, but it goes further by allowing users to associate one or more compilers with a given platform. The end result is a “fallback” system wherein users specify the preferred compiler and zero or more alternatives if the preferred compiler is not available. To achieve this flexibility, the Doozer++ `m4` macros must have no compiler-specific bits.

## Language Freedom

Language freedom is slightly more complicated than compiler freedom. As will be discussed more fully in the following section, Doozer++ separates its work into two pieces: configuration and compilation. During configuration, the tools for compilation are chosen; during compilation, the chosen tools are put to use.

The primary project building tool is GNU make. We chose GNU make initially because of its portability and because it offered useful features over most basic `make(1)` implementations. Other implementations, such as BSD make, offer even more features, but they are not as portable as GNU make.

Because Doozer++ uses GNU make, users must write makefiles. Doozer++ provides a number of modular makefile “stubs” that collect common functionality. In particular, these stubs provide support for compiling code written in Java, C, C++, and IDL. Adding support for other languages can be done within Doozer++ or within user-level makefiles.

## Separation of Tools and Tasks

The design of Doozer++ involves a distinct separation between the tools used and the tasks performed. As a result of this separation, there exists the possibility to extend Doozer++ to use tools other than Autoconf and GNU make. Moreover, different combinations are allowed. For example, we have found

that `make(1)` does not deal well with compiling Java code in general. A more appropriate tool for this task has come out of the Apache Jakarta Project [<http://jakarta.apache.org/>] called Ant [<http://jakarta.apache.org/ant/>]. Within Doozer++, we could make use of Ant instead of or in addition to GNU make to compile Java software. Nothing restricts users to Autoconf and/or GNU make.

The basic idea behind the separation is the following: detect and configure the static information needed to execute tools that compile a (potentially complex) software system. This leads to a two-step process: configuration and compilation. These steps are described in the following subsections.

## Configuration

During the configuration stage, we determine what tools are available that meet a given set of needs. Typically, this means finding suitable compilers and determining what options the compilers support. Of course, most projects have more complicated needs than this. For example, VR Juggler can be compiled with several different versions of GCC (2.95.3 through 3.1 as of this writing). When moving between platforms, the GCC C++ compiler will almost always be called `g++`, but it may not always be the same version<sup>2</sup>. For that reason, it may be necessary to perform several version-specific detection steps, including, but not limited to, the following:

- Whether a given option is required or even supported (for example, `-fexceptions` or `-LANG:<arg>`)
- What header files are available (`hash_map.h`, `hash_map`, `ext/hash_map`, or none of the preceding, for example)
- What libraries are needed for linking shared libraries or executables (`libdl`, `libposix4`, or `ws2_32.lib`, for example)

Moving beyond compiler-specific issues, it may be necessary to detect the installation of third-party libraries or programs. Often times, simply finding an installation is not sufficient. The version of the installation must also be checked to ensure compatibility with the user's project.

This all boils down to one thing: automation. The job of the configuration step is to automate as much as possible. In so doing, the code used to write the compilation can be simplified. In effect, the compilation step becomes a very generic process that is configured based on many platform-, compiler-, and site-specific details that cannot be detected easily by a tool designed for compiling.

## Compilation

As we just described in the preceding section, the compilation step should be a very generic process. Compiling software tends to be a very serialized or step-by-step process. Subsequent steps depending on proper completion of preceding steps. Nothing in particular about compiling software (or code generation in general) has to be tied to a specific set of tools. The process should depend more on the source code and what steps are necessary to generate the desired outcome.

While it is possible to construct build system software that does everything in the compilation step (a la Doozer proper), such systems tend to be very inflexible. Everything that could be provided more generically through the configuration phase must be defined statically in makefiles (or whatever specification file is being used to direct the build process) and in the code. For example, consider the following C++ code:

```
#if defined(HAVE_HASH_MAP)
```

---

<sup>2</sup>On a platform with multiple GCC installations, the executable names typically vary based on the version. For example, a FreeBSD 4.x installation with multiple GCC builds may have the executables `g++`, `g++30`, `g++31`, and `g++32` for versions 2.95.4, 3.0, 3.1, and 3.2 respectively. On RedHat Linux 7.2, `g++` is GCC 2.96 while `g++3` is GCC 3.0.4.

```
#include <hash_map>
#elif defined(HAVE_EXT_HASH_MAP)
#include <ext/hash_map>
#elif defined(HAVE_HASH_MAP_H)
#include <hash_map.h>
#else
#error "std::hash_map is not available with this compiler"
#endif
```

The code above is not tied to any specific compiler or any specific compiler version. Now, consider the following code that achieves the same result:

```
#if defined(__GNUC__)
# if __GNUC__ < 3 && __GNUC__ >= 2 && __GNUC_MINOR__ >= 95
#   include <hash_map>
# elif __GNUC__ >= 3
#   include <ext/hash_map>
# else
#   include <hash_map.h>
# endif
#elif defined(__MSVC_VER__)
# if __MSVC_VER__ >= 7
#   include <hash_map>
# else
#   error "std::hash_map is not available with this compiler"
# endif
#elif defined(__sgi__)
# include <hash_map>
#else
# error "std::hash_map is not available with this compiler"
#endif
```

The former is much shorter and hides all the platform- and compiler-specific details in the definition of the various `HAVE_*` symbols. The latter code is clearly much more complicated and only supports three compilers: GCC, Visual C++, and MIPSpro. (The latter example may also have inaccuracies. Think of it more as pseudo-code than something taken from real C++ code.) Certainly, the latter could be simplified by adding command-line options defined in platform- and compiler-specific *makefile stubs* (-`DHAVE_HASH_MAP` and so on), but such options would have to be provided for all possible cases. That method has obvious scalability problems, however.

Many other projects provide a variety of platform- and tool-specific makefile stubs that set up a build environment at the time of compilation. This inevitably leads to an ever growing number of stubs as the software becomes more portable or as external tools (especially compilers) evolve. For example, omniORB 3.0 (an excellent, freely available ORB implementation) has forty makefile stubs, four of which are for use on varying configurations Linux/i386. The upcoming omniORB 4.0, on the other hand, uses Autoconf to separate the platform-specific pieces from the generic, platform-independent compilation work.

## Further Developments

There are many long-term goals for Doozer++ that are beyond the scope of this document. As should be evidenced from the previous sections, however, some near-term goals include the use of project builder tools besides `make(1)`.

---

# Chapter 3. Goals of the VR Juggler Build System

The VR Juggler 2.0 build system is based on Doozer++, and thus the goals of Doozer++ extend into the VR Juggler build system. The VR Juggler build system has a much narrower scope than Doozer++, so it has some unique goals, and they are as follows:

- Centralize the complexity of the build code
- Minimize what users must remember to configure and compile the source
- Port quickly to new platforms

In the following sections, we discuss each of the above goals in more detail.

## Centralize Complexity

Due to the size of the VR Juggler 2.0 source base (nearly 500,000 lines of C++ and Java by one estimate), it is important that the complexity of compiling be centralized rather than spread out over the entire source tree. Centralizing the complexity allows most of the work to be done once, the results of which can then be reused by the rest of the build code. This is the key tenet for the other goals.

Centralization of complexity allows the VR Juggler build to follow the goal of Doozer++ wherein tools and tasks are separated based on what they do. The result is that each of the module's has an Autoconf- and Doozer++-based configure script that bears the brunt of the work needed to hide complexity. Then, each module has one or more “*glue*” *makefiles* that pull everything together. These glue makefiles typically direct a given module's build process so that steps occur in the correct order. Because they oversee the whole compilation process, they are often large files with many targets, each of which is responsible for a specific task. With a configure script and a glue makefile, all that remains is a listing of source files that must be compiled. The makefiles that list source files are often very, very short and easy for anyone to extend.

With this foundation, the complexity is separated thusly:

- Platform-specific work happens in a configure script where a flexible programming language is readily available
- Flow of execution during compilation is directed by a single, often long, glue makefile
- Source listings (for any language) are placed in short makefiles that are called by the glue makefile

Other steps have been taken to offload complexity or repeated tasks into centralized components. For example, the directory `juggler/macros` contains Autoconf/m4 macros that can be used in configure scripts. Most of the macros are used to detect and provide information about the various modules that comprise VR Juggler. There are a few helper macros that build on top of Doozer++ and common idioms we have used to simplify configure scripts further.

## Minimize Special Steps

As has been discussed at great length previously, it is important that users of the build system be freed

from remembering many command-line arguments or special steps that must be taken to build a given piece of software. With several modules to configure and build, there is a lot to know about compiling VR Juggler. Most modules have external dependencies, the paths to which must be specified using command-line arguments. Furthermore, some modules may be compiled differently based on configuration-time settings. The so-called “global build” manages all of this, and so it has the responsibility of reducing the amount of information users must remember.

To that end, the global build offers some features for remembering command-line arguments so that users do not have to. Many of the configure scripts for the individual modules are written such that the default values for arguments are reasonable enough that users rarely have to pass them. The combination of these two features usually allows the typical developer to run the global configure script with no arguments. While this is not always the case (for example, on Win32, the path to NSPR must be specified unless the user has done something special in his or her environment), it happens often enough to keep people reasonably happy.

## Port Quickly to New Platforms

VR Juggler as a whole is a cross-platform software system. Porting to new operating systems is a non-trivial task, and spending time porting the build system to new platforms is time that could be spent more effectively. Speaking less abstractly, before the source code can be ported to a new platform, the build system has to be capable of compiling the source code on the new platform. For that reason, it is important that the entire build system can be put to use right away so that the attention can be devoted to the more difficult task of porting C++ code.

The VR Juggler build system is limited in its portability by the portability of Doozer++. To its credit, Doozer++ is more portable than VR Juggler at this time, so the VR Juggler team still has some room to move. Even so, the VR Juggler build has its own quirks, and thus, people writing the code to build VR Juggler must always have portability in mind. For example, “BASH-isms” must never appear in VR Juggler configure scripts or makefiles. Most Linux distributions may use BASH for `/bin/sh`, but that certainly does not mean that all operating systems vendors follow that unfortunate trend.

In keeping with the Doozer++ goal of separating tools and tasks, the VR Juggler build system offers good portability by putting all of the platform- and software-specific pieces in Autoconf configure scripts. In so doing, the makefiles rarely, if ever, have to be modified when a new platform is added to the list. Furthermore, makefiles for GNU make lack sufficient programmatic constructs to provide developers with the ability to write tests that provide more than limited portability. While the Doozer software is relatively portable through its use of GNU make and platform-specific makefiles, it is missing the expressiveness of something based on Autoconf (or a similar tool). As discussed in the previous chapter, the use of `make(1)` alone for portability requires much effort to make hard-coded, platform- and site-specific makefiles.

---

## Part II. Design

“Three Rings for the Elven Kings under the sky, Seven for the Dwarf Lords in their halls of stone, Nine for Mortal Men doomed to die, One for the Dark Lord on his dark throne, In the Land of Mordor, Where the Shadows lie. One Ring to rule them all, One Ring to find them, One Ring to bring them all, And in the darkness bind them, In the Land of Mordor, Where the Shadows lie.”

--J.R.R. Tolkien, *The Lord of the Rings*, Volume 1, page 81

---

---

# Chapter 4. The “Global” Build

In this chapter, we present the design of the so-called “global” build. We cover the high-level aspects of the build system that ties together all the other build systems. In a sense, this is the one build system to rule them all.

## Follow the Rules

In order for the global build to work, the modules it wraps must follow certain rules. If a module does not comply with all the rules, there is no guarantee that it will be able to compile under all circumstances. In other words, “rogue” modules that only implement a few pieces of the puzzle quickly become the weak link in the chain, and module build system authors who want to live outside the rule set complicate matters for everyone else.

Because the VR Juggler build system is a complicated dance, there are many rules that must be followed. For example, certain targets must be defined so that recursive `make(1)` calls can proceed through the entire source tree. These targets include `'release'`, `'install-debug'`, `'links'`, and `'build-world'`. Other rules include restrictions on the names of files or directories, use of platform-specific conventions, the presence of a working `-config` script (see the section called “The `*-config` Scripts and the `*.m4` Macro Files”), and provisions for the detection of a usable installation. The full list of rules are provided in the following subsections.

## Required Targets

There are a number of targets required by the global build. Some of these were listed above. The following sub-subsections give a complete list of all targets that must be implemented by a module's glue makefile. The targets are grouped by the task they perform.

## Build Targets

The build targets have the job of building source code. What this means is up to the individual module. It may, for example, include the generation of source code using tools such as an *IDL compiler*, an *XSLT processor*, or a Java compiler. The targets need not do anything if, for whatever reason, the generic concept of building source code does not apply.

<code>build</code>	This target builds everything. It executes the first phase of the <code>'world'</code> target (i.e., only the build phase, not the install phase). Since it builds both debugging and optimized versions of a module without installing, it is useful for testing changes to the library code to ensure that it works in both the debugging and optimized cases. A profiled version of the module may also be built if the module build system supports that and the compiler can build such a version.
<code>buildworld</code>	This target is the same as <code>'build'</code> .
<code>debug</code>	Build only the debugging version of the module. If the target platform supports both types, static and dynamic versions are compiled. In other words, the module is built so that debugging symbols are turned on. It is the combination of <code>'dbg'</code> and <code>'dbg-dso'</code> (see below). <i>This is the default target and is what gets built if running <code>make(1)</code> with no arguments.</i>
<code>optim</code>	Build only the optimized version of the library binaries (both static and dynamic). This is built with no debugging symbols at all. It is the combination of <code>'opt'</code> and <code>'opt-dso'</code> .

profiled	Build only the profiled version of the library binaries (both static and dynamic). This capability is dependent on the compiler being used. Not all compilers support the process of generating profiled code, so this target may have no effect. Profiled libraries are built with debugging symbols. This target is the combination of 'prof' and 'prof-dso'.
dbg	Build only the <i>static</i> debugging version of the libraries. This does the same thing as 'debug' but does not compile the dynamic libraries.
dbg-dso	Build only the <i>dynamic</i> debugging version of the libraries. This does the same thing as 'debug' but does not compile the static libraries.
opt	Build only the <i>static</i> optimized version of the libraries. This does the same thing as 'optim' but does not compile the dynamic libraries.
opt-dso	Build only the <i>dynamic</i> optimized version of the libraries. This does the same thing as 'optim' but does not compile the static libraries.
prof	Build only the <i>static</i> profiled version of the libraries. This does the same thing as 'profiled' but does not compile the dynamic libraries.
prof-dso	Build only the <i>dynamic</i> profiled version of the libraries. This does the same thing as 'profiled' but does not compile the static libraries.

## Installation Targets

The installation targets set in motion the process of installing a module. As with build targets, what this means may vary from module to module. Each module is responsible for ensuring that its installation hierarchy exists before trying to copy files.

install	This is the complement to 'build' (described in the section called “Build Targets”), and in most cases, it is assumed that the build was performed before an installation is attempted. This target executes the second phase of the 'world' target. It performs a complete installation of debugging and optimized versions of a module. Installation of a profiled build will be performed if a profiled version was generated. Further, both the dynamic and static versions of a module will be installed if the target platform supports both. (This is of course assuming that the module builds one or more libraries.)
installworld	This target is the same as 'install'.
install-debug	Install only the debugging version of the module. If the module includes one or more libraries, both static and dynamic versions of the libraries are installed.
install-optim	Install only the optimized version of the module. If the module includes one or more libraries, both static and dynamic versions of the libraries are installed.
install-profiled	Install only the profiled version of the module. This may have no effect if the module build system does not support building profiled code or if the compiler cannot generate profiled code. If the module includes one or more libraries, both static and dynamic versions of the libraries are installed.

## Multi-Step Targets

There are a few multi-step targets required by the global build. Essentially, these targets perform builds and installations, though they do not necessarily build and installed exactly the same thing. They are intended to be used for making releases or for users who simply want a one-step build/install of a module.

world	<p>Clean up the build environment and then build and install everything using the default ABI and ISA. This is a simple target for those who just want to build and install the module as simply as possible. “Everything” in this case is the following:</p> <ul style="list-style-type: none"><li>• Debugging, optimized, and profiled versions of the library binaries</li><li>• Shared and static versions of the library binaries (if both are supported on the target platform)</li><li>• Header files</li><li>• Sample applications, test code, user tools, etc.</li><li>• Data files (sample config files, model files, whatever)</li></ul>
world-all-abi	<p>This is the same as the 'world' target except that it builds and installs <i>all possible</i> ABI and ISA combinations for the target platform. On IRIX, for example, this means that all combinations of N32, 64, mips3, and mips4 (debugging and optimized versions) are built and installed. Most platforms currently support only one ABI/ISA combination thus making this target the same as 'world'.</p>

### Note

As of this writing, the global build does not have this target. Some modules in the Juggler Project still do not support building multiple ABIs.

release	<p>This target is similar to 'world' except that the installation tree is suitable for redistribution. Extra files such as the change logs, the release notes, and the license files are installed. In addition, the tree is stamped with a build time to help track possible differences between two releases of the same version. (This has only occurred for one VR Juggler beta release, but it seems like a good idea to have the build time included with a distribution.)</p>
release-all-abi	<p>This is the same as the 'release' target except that it builds and installs <i>all possible</i> ABI and ISA combinations for the target platform. On IRIX, for example, this means that all combinations of N32, 64, mips3, and mips4 (debugging and optimized versions) are built and installed. Most platforms currently support only one ABI/ISA combination thus making this target the same as 'release'.</p>

### Note

As of this writing, the global build does not have this target. Some modules in the Juggler Project still do not support building multiple ABIs.

## Clean-Up Targets

There are three targets used to clean up the build environment. Each cleans the tree to a different degree. Of the following three, 'clean' and 'cleandepend' remove disjoint sets of files. The 'clobber' target performs at least the tasks of 'clean' and 'cleandepend'.

clean	Clean up everything in the build environment. This uses the 'clean' target defined by Doozer++ that is automatically included by all makefiles. The cleaning process is recursive just as the build process is. Each makefile may define which files are safe for cleaning, but generally core files, compiler-generated files, and object files are the only things removed during this process.
cleandepend	Clean up the automatically generated dependency files (the .d files in each directory). This method for cleaning up deletes only these files and nothing else—ever.
clobber	Clean up (clobber) the entire build environment except what was generated by <b>configure</b> . This runs the above clean-up targets and removes the object directory(ies) and lib directory(ies). Its purpose is to reset the build environment to its state just prior to running <b>configure</b> .

## Developer Targets

Finally, there are two targets that are relevant only to developers. These relate to the developer installation (see the section called “The Developer Installation”). One creates the developer installation, and the other removes it.

links	Set up the developer pseudo-installation environment.
clean-links	Remove the developer pseudo-installation environment.

## File and Directory Names

There are certain naming conventions for files and directories that must be followed in order to ensure consistency among all the modules in the Juggler Project. Not all of these relate directly to the build system, but the names may be influenced by the way the build system works.

## Autoconf-Generated Header File

All the modules (save one) make use of an Autoconf-generated header file that sets up #defines based on tests performed by the module's configure script. To avoid overlap or confusion, these files are named based on the module's C++ namespace. Furthermore, the header files must be generated within the module's unique header directory. For example, in Gadgeteer, the C++ namespace is `gadget`. Hence, the header file is `gadgetDefines.h`, and it is generated to the `gadget` directory.

The reason for the redundancy is to prevent user errors by avoiding ambiguities. Consider the following bit of code:

```
#include <defines.h>
#include <vpr/Sync/Mutex.h>
#include <vrj/Kernel/Kernel.h>
```

Now, for the sake of this example, assume that a user had both `-$VJ_BASE_DIR/include/vpr` and `-$VJ_BASE_DIR/include/vrj` on his or her command line. If both VPR and VR Juggler had a `defines.h` file, there would be no way to distinguish which is which. While this is a textbook case of operator error, the naming convention we use avoids this case entirely.

The problem arises because the generated header files do not include any other headers in the project. As a result, the generated headers are not tied as tightly to the directory structure as are the static headers. Hence, the above case is not so far-fetched. It could happen very easily with an inexperienced user.

## Module Configuration Header

Each module has a single header that includes the header file generated by running `configure`. The idea here is to have a single point where common actions are taken based on what comes in through the generated header. For example, based on platform settings, symbol export macros are defined in the module configuration header. This single header is then included by all the other files in the project.

The naming convention for the module configuration header is the same as that of the generated header except that the word “Config” is used instead of “Defines”. The reasoning for this convention is similar to that of the generated header, but in this case, at least one other file from the project is being included. Namely, the generated header file is always included on the first (non-comment) line of this header file. We decided long ago that the name `Config.h` was too common and needed an extra bit of uniqueness. Again, this is done to prevent user errors.

## Data Directory

In VR Juggler 2.0, the installation of multiple modules must be managed so that one module's (optional) extra data does not conflict with that of another module. All data must be installed into the directory `$(prefix)/share` (to use some `make(1)` notation). To prevent conflicts with other modules (and with other software that may already exist on the target machine), each module must name a project data directory (the variable `$(projdatadir)` is used to store this in the makefiles). In most cases, the unique directory should be the name of the project in lowercase letters with no spaces. For example, the directory for JCCL would be `$(prefix)/share/jccl`, and the directory for VR Juggler would be `$(prefix)/share/vrjuggler`.

## Installation Hierarchy

The structure of an installation hierarchy is fairly open, but there are several basic requirements. They are as follows:

- Headers go in `$(prefix)/include`. Ideally, a module will use a subdirectory of that for its own header files.
- User-accessible executables/scripts go in `$(prefix)/bin`.
- Libraries go in subdirectories of `$(prefix)/lib` (or a variant thereof depending on platform-specific conventions). More specifically, optimized libraries go in `$(prefix)/lib/opt`, debugging libraries go in `$(prefix)/lib/debug`, and profiled libraries go in `$(prefix)/lib/profiled`. There is further subdivision within those directories based on the binary format (ELF, a.out, etc.) and the instruction set architecture (i386, i686, mips4, sparc, etc.). To make things more convenient for users, symlinks to (or copies of depending on the host platform) libraries should be made in `$(prefix)/lib`. For full releases, we make symlinks to the optimized libraries. In the developer installation, we make symlinks to the debugging libraries. Typically, profiled libraries will be named differently than their non-profiled counterparts, so symlinks to those can be made along side the non-profiled versions.
- Project data files and sample code goes in `$(prefix)/share/<project-name>`. The use of the `<project-name>` subdirectory is to avoid conflicts with existing software.

## `configure.pl`

The script `configure.pl` is a Perl script written to act as part of a build wrapper around an arbitrary collection of software modules. The modules are linked through some sort of (conceptual) *dependency graph*, in this case specified by a simple configuration file (see the section called “Build Configuration

File”). `configure.pl` reads the configuration file and proceeds to configure the individual modules in an order that satisfies the dependencies. Along the way, environment variables are set or extended so that each subsequent module is configured with the correct settings. The processes of managing the dependencies and performing the configuration are the topics of this section.

## Build Configuration File

Before explaining how `configure.pl` manages its dependencies, it will be helpful to understand the role played by the configuration file from which the dependencies are read. In the VR Juggler build system, the file is called `juggler.cfg`, and it is located in the top-level source directory. At a very high level, the configuration file defines one or more modules that must be configured and compiled. The modules may be independent of each other, or they may form a dependency graph. In the latter case, one module states that it depends on one or more other modules. The following code block shows an example of this:

```
module VPR
{
    external;
    modules/vapor;
}

module Tweek
{
    depend VPR;
    modules/tweek;
}
```



- 1** These lines declare two modules named “VPR” and “Tweek” respectively.
- 2** These lines list directories upon which the containing module depends. Each of these directories must contain a script called `configure` that can be executed. This method of listing dependencies requires explicit paths, and it helps greatly if those dependencies exist within the local source tree.
- 3** This line indicates that the Tweek module depends on the VPR module. Here, note that `depend` is a keyword with special significance. In effect, the VPR module is included inside the Tweek module definition so that it picks up all of VPR’s dependencies.

Going deeper into the module definition, we find that environment variables can be set with each directory listing using a comma-separated list. These variables provide extra information about the configuration environment after the `configure` script has completed successfully. To illustrate this, we extend the above example as follows:

```
module VPR
{
    external;
    modules/vapor: VPR_CONFIG=vpr-config, VPR_BASE_DIR=instlinks;
}

module Tweek
{
    depend VPR;
    modules/tweek: TWEAK_CONFIG=tweek-config, TWEAK_BASE_DIR=instlinks;
```

- 1** As before, these lines list directories upon which the containing module depends. This time, we have added environment variable settings for the variables `$VPR_CONFIG`, `$VPR_BASE_DIR`, `$TWEAK_CONFIG`, and `$TWEAK_BASE_DIR`.

While any environment variable can be set in the configuration file, those shown above have special significance<sup>1</sup>. Those variables ending in `_CONFIG` set the corresponding environment variable to include the full path to the named file (despite the fact that the full path is not given in the assignment). The path is constructed using the associated directory dependency. This directory is also added to the script's execution path. This extra little bit is needed so that a given `-config` script can be executed by another `-config` script if necessary. (The details about why all of this is necessary are discussed in the section called “The `*-config` Scripts and the `*.m4` Macro Files”.)

Those variables ending in `_BASE_DIR` define the installation directory for the given module. In the VR Juggler build system, this is necessary for dependent modules to find the headers and libraries they need to compile. (Again, more information about this is given in the section called “The `*-config` Scripts and the `*.m4` Macro Files”.) Once again, the value being assigned has special significance. If the value is the token `instlinks`, it is taken to mean that the *full path* to the installed module is in a directory relative to the current directory called `instlinks`. Any other value is used verbatim as the value of the environment variable.

## Dependency Management

Dependencies within modules are maintained using a simple Perl data structure in the `JugglerModule` class. Parsing the configuration file results in instantiation of this data structure. There is one such instance for each module defined. Each instance contains an array of `ModuleDependency` objects. Steps are taken to ensure that there is no duplication of dependencies within a single `JugglerModule` instance.

## The `*-config` Scripts and the `*.m4` Macro Files

The various `*-config` scripts (`vpr-config`, `tweek-config`, `vrjuggler-config`, etc.) play a vital role in the design of the global build. Unfortunately, this is also where the global build gets complex. Here, code that is intended for use by users of VR Juggler and associated modules is put to use by the code that compiles everything. It makes use of a strict set of behaviors wherein various environment settings and command-line options form a hierarchy of preferences and fallbacks. If anything goes wrong with a user's configuration process, it is almost always related to a `.m4` file or a `-config` script giving unexpected results because of a misused command-line option or a “dirty” environment.

## A Typical `-config` Script

We will now examine a typical `-config` script. We will not focus on any script in particular, but instead, we will describe the fundamental concepts and requirements shared by all implementations. Readers interested in implementations to use as references should consider the following:

- `vpr-config`: This is the most basic `-config` script. It has no external dependencies, and it only deals with one library.
- `tweek-config`: This script depends on `vpr-config` for proper execution, and it deals with some interesting special cases. Namely, it must be able to inform callers about information relating to C++, Java, and IDL. This script deals with one C++ library and multiple Java libraries distributed as *JAR*

---

<sup>1</sup>The fact that any variable can be set but that some are treated as special cases is a deficiency in the design of `configure.pl`. The current use was put together out of necessity to provide the script with extra information needed for proper execution of each module's Autoconf-generated configure script.

*files.*

- `vrjuggler-config`: This script is interesting because it has the most dependencies (it depends on `vpr-config`, `tweek-config`, `jccl-config`, `gadgeteer-config`, and `sonix-config`) and because it deals with multiple libraries (`libJuggler`, `libJuggler_ogl`, and `libJuggler_pf`).

To proceed with the abstract discussion, we will now describe the job that must be performed by all `-config` scripts. We then explain how external dependencies are managed. We conclude with a discussion of how a `-config` script is generated as part of the configuration process.

## The Job

The job of any `-config` script is simple: provide the information needed to compile against the associated library. This information can work in the context of building a higher level library or an application. The basic information that must be provided is as follows:

- Module version number
- C++ compiler flags including header paths and compiler-specific options such as `-fexceptions` or `-LANG:std`
- C++ linker flags separated into two categories:
  1. The basic list of libraries that are distributed with the module in question
  2. The complete list of external dependencies needed to link an application
- Static linking options
- Profiled library linking options (if profiled libraries are available)

Other information may be provided as necessary (see `tweek-config` and `jccl-config`, for example).

## External Dependencies

In the Juggler Project, we use the `-config` scripts in an interesting manner. A given script, say `gadgeteer-config`, will call all the `-config` scripts of the modules on which it depends. The collected output is compressed and returned to the user. In this way, we avoid trying to manage all the dependency information in every module. Instead, we rely on each module to report its information correctly. Then, the highest level module only has to collect it and print it out.

The key to this functionality is that all the `-config` scripts can be found in the user's path. Furthermore, because our `-config` scripts are written in Perl, we have easy access to the path used to invoke each script. (This is actually a side effect of the scripts being in the user's path, and the path information would be available regardless of the scripting language. Perl just makes it easy to extract and operate on the given information.)

The dependencies only come into play for certain information requests. Those requests are the following:

- C++ flags (`--cxxflags`)

- Include paths (`--include`)
- Basic libraries (`--libs`)
- External dependency libraries (`--extra-libs`)

Each of the above iteratively calls the `-config` script(s) from the dependency module(s) using at least a subset of the arguments specified by the user on the command line. One additional argument is given that reduces the amount of output: `--min`. This causes each script to print out only the minimal information needed for compiling. The motivation for doing this is to keep the compile lines short whenever possible.

## Script Generation

Finally, we explain the last detail relating to all `-config` scripts used in the Juggler Project: script generation. As part of the global build, all these scripts must be generated at configuration time. This is necessary for each subsequent module to be configured correctly. That means that each module's configure script is making use of “external” `-config` scripts to get command-line arguments and version information. This allows the use of pre-existing installations of dependencies. For example, Gadgeteer depends on JCCL and VPR. Normally, it would satisfy those dependencies using source from the same tree, but a user may already have JCCL and VPR built and installed. By setting up his or her path correctly, `vpr-config` and `jccl-config` can be found, and Gadgeteer can be built using the existing installations.

As of this writing, there are no hard and fast rules regarding script generation other than the fact that they must be generated as part of module configuration. The prevailing convention within a given `configure.in` is to make separate *substitution variables* used only in the `-config.in` template file. That is to say that these variables are separate from those used in template makefiles and other `.in` templates. The separation is done through syntax alone. Those variables that are substituted in the generation of a `-config` script are spelled using lowercase letters exclusively. Other substitution variables are spelled using all uppercase letters.

Again, this is not a hard and fast rule. Indeed, some variables are shared between all files (for example, `$USE_GCC` or `$MAJOR_VERSION`). In general, such exceptions are allowed because there is no difference in usage between a `-config` script and a generated makefile. The important distinctions arise with compiler and linker flags. In particular, there are some flags that should only be used in the process of building a given module but should not be exposed to users. An example is the MIPSpro `-woff` flag that is used to disable compile-time warnings. Exporting this option would force users to disable the same warnings whether they want to or not. Essentially, it is up to the module build system author to use good judgement when deciding what to export and what to use internally.

To summarize, the separation of the substitution variables is done for two reasons:

1. To manage potential differences in semantics between module compilation and module use.
2. To make it clear to readers of the relevant files which variables are used for which purpose.

To date, this mechanism has worked well (at least for those few who know about it). In one case, failure to follow this convention caused compilation of a module to break because a linker variable was serving double duty. This variable was setting internal linker options and forcing the use of those same options for external code.

## A Typical .m4 File

With an understanding of what information a `-config` file provides, we can move on to the m4 macros that make use of those files at configuration time. Each module in the Juggler Project has a correspond-

ing m4 macro suitable for use within an Autoconf `configure.in` file. For example, Gadgeteer has a macro called `GADGETEER_PATH`. This macro is used in VR Juggler's `configure.in` to find a usable Gadgeteer installation.

The basic concept behind all these macros is the same: provide a way for a configure script to detect a usable installation. The way this is done is fairly straightforward. The basic step-by-step process is as follows:

1. Find the module's `-config` script. If the script cannot be found, execute the user-specified failure steps and “return”<sup>2</sup>.
2. Using the `-config` script, get the version of the installation and compare it against the user-specified minimum required version.
3. If the version is sufficient, execute the user-specified success steps and set variables for compiler and linker arguments using the `-config` script. If the version comparison fails, execute the user-specified failure steps and “return”.

We now proceed into the details of a typical `.m4` file. First, we will cover the mechanism used to deal with paths to installations. Then, we explain what variables must be set by a module's m4 macro.

## Path Setting Preferences

The most complex part about understanding the Juggler Project `.m4` files is the management of path setting preferences. The path in question is the path to the installed module. The installation may be on the local file system, or it may exist as a developer pseudo-installation (refer to the section called “The Developer Installation” for more information on that topic). The specification of that path by the user is where we now direct our interest. The following list gives the path setting preferences in order of decreasing preference (i.e., the first has highest precedence, and the last has lowest precedence):

1. The `_CONFIG` environment variable which gives the full path to the module's `-config` script. The name of the environment depends on the specific `.m4` file. For example, `vpr.m4` checks for `$VPR_CONFIG`. In general, the environment variable name should match the name of the `-config` script except in capitalization and the use of an underscore (`_`) instead of a hyphen (`-`).
2. The command-line argument `--with-<module>-exec-prefix` which specifies the directory containing the `-config` script. Here, the string `<module>` depends on the way the `.m4` file is written to declare accepted command-line arguments.
3. The command-line argument `--with-<module>-prefix` which specifies the directory containing the full module installation. The named directory must contain a `bin` subdirectory, and this subdirectory must contain the module's `-config` script.
4. The module's `_BASE_DIR` environment variable. The directory named by the environment variable must contain a `bin` subdirectory that in turn contains the module's `-config` script.
5. The user's path which must include the directory that contains the module's `-config` script.

The magic that happens in `configure.pl` is a result of setting the appropriate `_CONFIG` environment variable and extending the path to include the various module directories in the build tree. Through the combination of these steps, a given `-config` script is found using the `_CONFIG` environment variable, and any other `-config` scripts it needs are found using the path. For each module that is built, the environment in which `configure.pl` runs is extended. Refer back to the section called “Build Configuration File” for information about how the necessary file and environment variable names are provided to `con-`

---

<sup>2</sup>Actually, these macros do not return because the code is inlined. Moreover, the user-specified failure steps may include halting the configure script and exiting with failure status.

## Variables Defined

Upon successful completion of step 3 (see above), there are several shell variables defined for the calling code to use. Typically, these variables are concatenated with other variables to form the full set of options passed to the compiler and the linker. In order to maintain consistency across all the `.m4` files, it is important to know what variables must be defined and why.

First, the variables can be separated into two broad categories: minimal and maximal. The basic idea with minimal versus maximal flags is to allow the user some flexibility in composing the full command-line options. The minimal variables provide only the minimum amount of information needed for compiling. No dependency information is included. For example, such settings usually include one or two header path extensions (`-I` options) and mandatory compiler flags such as `-LANG:std`. The maximal variables, on the other hand, include all the information needed for compiling including dependency data. More concretely, the minimal C++ compiler flags for Gadgeteer would give the header path for the Gadgeteer headers and any mandatory compiler flags. The maximal C++ compiler flags would include the minimal information as well as flags relating to JCCL, Tweek, and VPR (in that order). The same would be true for linker flags. To summarize, the minimal variables must be mixed with other minimal variables if the module has dependencies; the maximal variables can stand on their own.

Within the minimal and maximal categories, there are two more categories: compiler flags and linker flags. Based on the discussion above, this distinction should be fairly obvious. This distinction is made to deal with platforms where the compiler is not used to perform the link stage. Since some compilers can call the linker as necessary, *linking* flags<sup>3</sup> suitable for use with the compiler are provided. The distinction is made by the variable name. Linking flags that can be passed to the compiler have `_CC_` in their name; linking flags for the linker have `_LD_` in their name.

Continuing with the subdivision, the linking flags are divided into two categories: static and dynamic. This is done to allow static linking of one or more libraries instead of dynamic linking. The usual default would be dynamic linking, and the variable names reflect that. Those variables that contain static linking flags have `_STATIC_` in their names.

## The Developer Installation

At various times in the history of the VR Juggler project, the developer “pseudo-installation” has been a topic of great controversy. Questions (or arguments, depending on your perspective) regarding its usefulness and its differences from a release installation come up repeatedly. The reason for its existence is quite simple: to simplify the lives of developers. The differences between the developer installation are also fairly simple. In a nutshell, a developer installation uses debugging libraries by default and links applications statically. A release installation, on the other hand, uses optimized libraries by default and links applications dynamically. The reasoning behind this is a little more difficult to nail down, and for that reason, we will say that it is beyond the scope of the document.

To satisfy the only goal of the developer installation (simplification of developers' lives), the developer installation must act exactly like a release installation, but it must be inside the build tree. The developer installation is created automatically as part of the build process, and ideally, its construction is faster than that of a full release installation. In any case, if all goes well, a developer can treat this pseudo-installation as if it were a real installation for the purposes of running tests.

Prior to early August 2002, the developer installation was created separately from a release installation. This was done through the use of symlinks on UNIX-based platforms and file copies on Win32. Since August 2002, the developer installation still uses symlinks or file copies in the same manner, but there is no longer a separation between creation of the developer installation and the release installation. In other words, the release installation is used to make the developer installation, but it is directed to install into

---

<sup>3</sup>Note the distinction between *linker* flags (flags for the linker program) and *linking* flags (flags used to link object files).

the build tree.

The decision to use symlinks or file copies is based on the host platform and on the use of the 'links' target that every module must define. Using a custom Perl script, **bsd-install.pl**, written to be fully compatible with BSD **install(1)**, symlinks may be created instead of using file copies. (The **bsd-install.pl** script comes with Doozer++ 1.5 and beyond.) Within the script, a test is performed to determine if the host platform is a Win32 system. If so, copies are always used because there are no symlinks on Win32 file systems. All of these decisions had been made in each module's build system, but they have since been offloaded into **bsd-install.pl**. This is in keeping with the Doozer++ goal of centralizing complexity.

## Future Goals

At one time, there was a long-term goal for this global build script, or “project builder”. In conjunction with **cvs-gather.pl**, a semi-arbitrary collection of software packages would be downloaded and compiled. The commonality between them would be the use of configure scripts that would be invoked by the project builder. The dependencies would be specified through some configuration file, possibly written in XML, that would be constructed on the fly based on the **cvs-gather.pl** dependency file.

These lofty plans have not materialized, and it is unclear whether the need for such a tool still exists. Nonetheless, the idea of a highly generalized project builder played a key role in the way that **configure.pl** was written. In particular, its highly generic nature was motivated by the potential for downloading arbitrary source code and running a configure script. The makefile generated by **configure.pl** initially followed this goal, too, but it has since had its scope narrowed to deal with the specific conventions of the VR Juggler project.

---

# Chapter 5. The Documentation Build

The documentation for the individual modules is maintained separately from the source code build. This is done primarily because writing code to DocBook and related software would take too much time and gain us very little. For the most part, the Juggler Project documentation is intended for posting on the website. As such, the build environment is much more controlled than one that is provided for (easy?) use by the general public. The documentation build is still configurable, but in a different, less automatic method than what is used to build the source tree. This chapter explains the rules of the documentation build and how to configure it for use outside the VRAC lab.

## The Rules

Any module with DocBook-based documentation can be added to the documentation build. As with the source code build, there are rules that must be followed. With this smaller, less complicated build system, there are fewer things to deal with.

## Required Makefile Targets

Within this subsection, we present the full list of targets that must be implemented for correct operation within the documentation build. As a reference, refer to the file `juggler/Makefile.docs`. This list, containing all of two targets, is as follows:

<code>docs</code>	This target builds the documentation. It is up to each makefile to determine how much is built. In general, the makefiles are set up to generate HTML, PDF, and PostScript. This output may come from DocBook XML source or from output generated by Doxygen. Other tools may be used, but these two are the favored generators at this time.
<code>install-docs</code>	This installs the documentation built by the <code>'docs'</code> target. By default, the documentation installation uses <code>\$HOME/public_html/jugglerweb</code> as the base prefix. This truly reflects the intended use of this build system. (To redirect the installed output, simply change the setting for <code>\$(webroot)</code> in <code>juggler/Makefile.docs</code> .)

Targets such as `'clean'` and `'clobber'` are needed as well, but those makefiles that include `docbook.mk` get them for free. If a documentation generating makefile is not using `docbook.mk`, it must make its own `'clean'` and `'clobber'` targets.

## Installation Hierarchy

The individual documentation building makefiles are expected to behave when installing the generated documentation. That is, they should install to a subdirectory of `$(webroot)` that reflects the appropriate project. In many cases, the installation hierarchy should also reflect the version of the software against which the documentation was written.

Each makefile is responsible for creating its own installation hierarchy and for installing any related, external files. The documentation build offers some automation to help with this, but its abilities are limited. At this time, the documentation build can be directed to install the image files that come with the DocBook style sheets and any local images that the documentation needs.

## Configuration and Customization

At this time, most of the documentation in the Juggler Project is written using DocBook. Because of that, the settings for building documents from DocBook files are centralized in the file `juggler/doc/docbook.mk`. This file is parameterized to the extreme so that including makefiles can override its default settings easily. Usually, makefiles that include `docbook.mk` direct the build to use OpenJade, Saxon, or Xalan to process the DocBook input and PassiveTeX, FOP, or XEP to create PDF files. Further configuration can be done that chooses different versions or installations of the DocBook style sheets, Saxon, Xalan, FOP, etc. The following is the makefile used to generate HTML and PDF versions of this document:

```

default: html

docs: html chunk-html pdf

install-docs: install-html install-chunk-html install-pdf

include build.system

XML_FILES= $(NAME).xml
HTML_FILES= $(NAME).html
PDF_FILES= $(NAME).pdf
XSL_TOOL= Saxon

# Put these in!! Together, they make up the installation prefix.
webroot= $(HOME)/public_html/jugglerweb

indir= docs/juggler.build.system

index= $(webroot)/$(instdir)

HTML_FILES= $(webroot)/base_style.css

PDF_DB_IMAGES= 1

$(NAME).html: $(NAME).xml
$(NAME).pdf: $(NAME).xml $(NAME).fo

include ../docbook.mk

```

These are the required targets that all makefiles in the documentation build must have. Refer to the section called “Required Makefile Targets” for more information on these targets.

This variable lists the base name of the source document. Since the generated documents differ only in final extension, this variable is used internally as the basis for the name of the source file and the generated documents. If there are multiple source documents, multiple variables (such as `$(NAME1)`, `$(NAME2)`, etc.) would be used.

This variable lists all the DocBook XML source files that will be used as input to the XSLT processor. Each source file should have corresponding output files, also listed in this makefile.

The source document can be rendered as HTML, and so this variable is used to list the output file.

All HTML output files must be listed in this variable. The names must reflect the non-chunked

output file names. Chunked HTML output is handled separately since the file names cannot be listed easily in this context.

5 Similar to `$(HTML_FILES)`, this lists the PDF file(s) that can be rendered from the XML input. This has the same semantics as `$(HTML_FILES)`.

6 The variable `$(XSLT_TOOL)` is used to tell `docbook.mk` which XSLT processor should be used. The default tool is Xalan, but Saxon can be used instead, as shown above.

7 These variables deal with the installation process. The critical variables are `$(webroot)` and `$(prefix)`. The variable `$(webroot)` will be overridden by the calling makefile (the top-level documentation makefile) to provide an alternate base directory as necessary. Setting it here provides a good default value to use when writing documentation and testing rendering. The `$(prefix)` variable is what tells `docbook.mk` where the document(s) will be installed. It must be set in order for an installation to succeed. Finally, `$(INSTALL_FILES)` provides any extra files that must be installed. In this case, we want to copy over the common stylesheet used by the rest of the VR Juggler website.

8 Defining the variable `$(NEED_DB_IMAGES)` informs `docbook.mk` that the DocBook images are needed by rendered versions of the source document(s). Symlinks will be created as part of the rendering process, and the images directory will be copied over during installation.

9 These lines simply list dependencies as a way to help `make(1)` in doing its job effectively. Here, the `$(NAME)` variable again plays a useful role.

10 Finally, the last line (and it should always be the last line) includes the `docbook.mk` makefile stub. All of the settings above have an effect on the way that the targets defined in `docbook.mk` behave.

---

## Part III. Extension

At this point they came in sight of thirty forty windmills that there are on plain, and as soon as Don Quixote saw them he said to his squire, "Fortune is arranging matters for us better than we could have shaped our desires ourselves, for look there, friend Sancho Panza, where thirty or more monstrous giants present themselves, all of whom I mean to engage in battle and slay, and with whose spoils we shall begin to make our fortunes; for this is righteous warfare, and it is God's good service to sweep so evil a breed from off the face of the earth."

--Miguel de Cervantes, Don Quixote (English translation), opening of Chapter VIII

In this part, we discuss the topic that seems most fearsome to the majority of Juggler team members: build system extension. While this does not have to be a complex topic, most people shy away from it because they do not want to spend time trying to understand any given build system. Whether anyone will even bother to read this far into this book is debatable, and this is only the second attempt in four years to write this documentation. Perhaps this time, it will be worth the effort.

---

---

# Chapter 6. Extending the Global Configuration Script

Most extensions to `configure.pl` can be localized to the `JugglerConfigure.pm` module file. That module contains the configuration file parsing code and two data structures (`JugglerModule` and `ModuleDependency`) for managing the data that is parsed. In `configure.pl`, most of the code relates to execution of the configure scripts in the individual modules. This includes running the configuration scripts to configure a build tree and running them to collect, reformat, and output usage information (the output normally printed from passing `--help`). In this chapter, we examine both files separately and explain how to extend them.

## JugglerConfigure.pm

This file contains three Perl packages: `JugglerConfigure`, `JugglerModule`, and `ModuleDependency`. The first has a single procedure, `parseConfigFile()`, that parses the user-specified configuration (`juggler.cfg` in most cases). The latter two define data structures (Perl classes) used to manage individual modules defined in the configuration file and the dependencies of each module. We will discuss each package in turn.

## JugglerConfigure

As stated above, this package contains a single subroutine: `parseConfigFile()`. This parses a user-specified configuration file in the format described in the section called “Build Configuration File”. If the config file is parsed correctly, a hash named `%MODULES` is returned to the caller. This hash, indexed by module name, provides references to `JugglerModule` instances.

The parsing code is fairly straightforward. There are only three keywords in the configuration file format: `module`, `depend`, and `Default`. The basic concepts used in the format are similar to those for the C grammar. In other words, curly braces denote blocks, semi-colons denote the end of a declaration, and all code is case-sensitive. The following provides the grammar for the configuration file:

```
DOCUMENT ::= COMMENT DEFAULT MODULE
DEFAULT  ::= 'Default:' NAME | #
MODULE   ::= NAME '{' COMMENT DEPENDENCY '}' COMMENT MODULE | #
NAME     ::= ([A-Za-z0-9_][A-Za-z0-9_]* + #)[A-Za-z0-9_]
DEPENDENCY ::= DIR ';' DEPENDENCY | MODDEP ';' DEPENDENCY | #
DIR      ::= <valid-path> | <valid-path> ':' ENVVAR
ENVVAR   ::= NAME '=' <text> | NAME '=' <text> ',' ENVVAR
MODDEP   ::= 'depend' NAME
COMMENT  ::= '#' <text> | '///' <text> | '/*' <text> '*/' | #
```

The structure of this grammar should be reflected in the parsing code. Differences between strict regular expressions (used above, more or less) and Perl regular expressions may muddy the correlation.

## JugglerModule

The `JugglerModule` package defines a class containing two “data members”: `name` and `deps`. These are actually keys of the blessed hash returned by the `new()` subroutine. The `name` member is just a string that contains the unique name of the module. The `deps` member is more interesting. Indeed, the main purpose of the `JugglerModule` package is to maintain the list of dependencies.

The `deps` member is a reference to an array of `ModuleDependency` objects (see the section called “ModuleDependency”). An array is used to preserve the ordering read from the configuration file.

All dependencies given in the configuration file appear in the array. This includes those “inherited” from another module through the use of the `depend` keyword.

Duplication of dependencies is avoided by performing a check whenever a new dependency is added using `addDependency()`. The `hasDependency()` method encapsulates the process of making the check. The dependencies of two modules can be merged using the `addDependencies()` method.

## ModuleDependency

The `ModuleDependency` package defines a class containing two “data members”: `path` and `env`. The first is a simple string, and the second is a reference to a hash where the keys are environment variable names. The values for these are extracted from the `DIR` rule in the above grammar. More specifically, the terminal `<valid-path>` in `DIR` rule corresponds to the `path` member, and the `ENVVAR` rule defines the hash of environment variable mappings.

Access to a given `JugglerModule` object's dependencies happens entirely through the API of that class. This class is provided as a convenient abstraction around the information provided through each `DIR` parser rule expansion.

## configure.pl

This script pulls everything together. After calling `JugglerConfigure::parseConfigFile()`, the returned hash (see the section called “`JugglerConfigure`”) is used to perform all the relevant operations including, but not limited to, invoking each `Autoconf`-based configure script in the correct order. As a whole, the work done by **configure.pl** is fairly straightforward, regardless of the arguments passed on the command line. In this section, we will examine the four main aspects of this script's functionality:

1. Command-line argument handling
2. Help output
3. Build configuration
4. File regeneration

We will present each in turn with the goal that understanding these pieces of functionality, extensions to **configure.pl** will be easier to perform. Of these points, #3 will require the most discussion since that is the main reason for **configure.pl**'s existence.

## Command-Line Argument Handling

The handling of command-line arguments in **configure.pl** is rather unique. To function correctly as a wrapper around multiple `Autoconf`-based build systems, this script must be capable of passing the correct arguments to the individual configure scripts. In other words, when the user executes the following:

```
configure.pl --module=VPR --enable-subsystem=NSPR --with-nspr=$HOME/nspr-4.2
```

the argument `--module` must be handled by **configure.pl** while the other two are passed through to `VPR`'s configure script. Luckily, the Perl module `Getopt::Long` provides a mechanism for doing just that. The module is configured to allow unrecognized arguments to be passed through so that, after the execution of `GetOptions()`, `@ARGV` will contain what `GetOptions()` did not handle. When the time comes to execute a module's configure script, `@ARGV` is passed to the script as normal arguments.

This usage of `Getopt::Long` is not without a drawback or two. Allowing the pass through of unrecognized arguments means that `GetOptions()` is not performing robust error checking on user input. What if `--module` were spelled as `--mdoule` accidentally? `GetOptions()` would not recognize it, and the execution of **configure.pl** would not proceed the way the user had intended. Because of this, it is important for users to be especially careful with the options they give when running **configure.pl**.

## Help Output

Two types of output can be printed by **configure.pl**: usage for **configure.pl** and usage for all the known configure scripts. The first is done quite simply. The `Pod::Usage` module is used to render the internal *POD* as usage output<sup>1</sup>. This output is printed when the user passes `--help` on the command line.

Usage information for the known configure scripts can be printed using the `--all-help` argument. This is provided so that users have a way to get a complete listing of all the command-line arguments that may be passed when running **configure.pl**. (The exact details of how this works were described in the previous subsection.) Only the relevant output is actually printed when this argument is passed. The meaning of “relevant” is tied to the module the user is configuring. For example, with an unmodified `juggler.cfg`, the default module is “VRJuggler” (sic). If the `--module` option is not specified in conjunction with `--all-help`, then the usage information for the VR Juggler configure script and the use information for the configure scripts of all of its dependencies are printed. If a different module is specified, the output will change based on the alternate module and its dependencies.

To print this information, each configure script is run by **configure.pl** with the `--help` command-line option. The output is collected and processed so that the arguments are printed to the screen in an orderly fashion. Duplicated output is discarded. The process of presenting the output to the user is very important because there exists the potential for a large amount of information.

## Build Configuration

Configuring a build tree involves several steps, and they are as follows:

1. Load default configure script arguments (if any exist)
2. Generate the **reconf** helper script
3. Run the appropriate configure scripts in the right order
4. Generate the top-level makefile that invokes targets in each module's glue makefile

Each of these steps will be discussed in turn in the following sub-subsections.

## Load Default Arguments

To simplify the execution of **configure.pl**, users may write a default arguments file or a Perl plug-in that provides default arguments. In either case, the idea is that arguments for Autoconf-generated configure scripts can be collected into reusable platform- and site-specified pieces so that they do not have to be passed every time **configure.pl** is run. The preferred method for specifying default arguments is the Perl plug-in, but for users who do not know Perl, a simple configuration file can be used instead. Here, we will only describe the process of loading these files, not how to use them.

To load the Perl plug-in, the arguments file is treated as loadable Perl code. (Indeed, the requirements of

---

<sup>1</sup>Use of this very handy Perl module has actually been a source of problems. In particular, IRIX 6.5 ships with a version of Perl that does not contain this module. For that reason, Perl 5.005 or newer is required to run **configure.pl** and some other Perl scripts used by the build system.

using the Perl plug-in state that it must be valid Perl code with a subroutine that can be called by another package.) Once the plug-in is loaded into memory, its subroutine is executed. The subroutine must return an array of command-line arguments that can be merged with `@ARGV`. It is the responsibility of the plug-in to inform **configure.pl** of its executable subroutine by assigning a subroutine reference to a global variable in **configure.pl**'s scope. In **configure.pl**, this whole process involves roughly four lines of code, thus making it very attractive from the standpoint of maintaining **configure.pl**.

Loading the simple configuration file requires more effort because the file format must be parsed. In **configure.pl**, the parsing code for this file format is found in `loadDefaultArgs()`. To summarize the file format, platform-specific blocks are defined, and within a block, each line gives a configure script argument. The platform-specific blocks are named according to platform, though they can be named with user-defined identifiers when used in conjunction with the `--os` command-line option. Parsing the file simply means reading the right block(s) and adding each line to `@ARGV`.

## Generate reconfig

Generating the **reconfig** helper script is quite simple. The arguments passed on the command line are simply saved to the `reconfig` file. In addition, commands to remove any Autoconf cache files (named `config.cache`) are added. Finally, the **reconfig** file is made executable.

## Run the Configure Scripts

This step is where **configure.pl** works the hardest. Ultimately, what must happen is that a given module's configure script is executed, and if it exits without failing, the execution environment of **configure.pl** is extended to reflect new information. This new information comes from the environment variable settings in the original configuration file. The details of how this works are a little bit more complicated, however.

The real goal of **configure.pl** is to run the configure script of the module the user wants to configure and build. Along the way, it may have to run configure scripts for one or more dependencies. Dependency information has already been covered in great detail, and we will not repeat it here. To ensure that the dependencies are configured in the correct order, the array of dependencies within the `JugglerModule` object is requested. Each dependency's configure script is run, and the execution environment is extended accordingly. The last dependency of a module is the actual module the user wants to configure and build. Once its configure script completes successfully, the top-level makefile can at last be generated.

The environment variables that are set reflect the delicate balance between **configure.pl**, **configure.pl**'s configuration file, the `*-config` scripts, and the `*.m4` files (see the section called "The `*-config` Scripts and the `*.m4` Macro Files"). Based on the rules set forth by the `*-config` scripts and the `*.m4` files, the execution environment must be set up so that each subsequent configure script can get information about any dependencies it has.

## Generate the Top-Level Makefile

Generating the top-level makefile is relatively easy (it is mostly search-and-replace operations), but interestingly, this is one of the places where **configure.pl** falls short of acting like an Autoconf-based configure script. A template makefile (`juggler/Makefile.in`) is always the only file generated by **configure.pl** (aside from the **reconfig** script). There is no way to specify other files. Even if there were, **configure.pl** does not have easy access to the results of all the tests performed by the various configure scripts.

As a result of these limitations, there are only a few strings that can be expanded in the generated makefile. At this time, only two variables can be expanded: `@JUGGLER_PROJECTS@` and `@topdir@`. The expansion of the first depends on the configured module and its dependencies; the expansion of the second is simply the directory where **configure.pl** was run.

## File Regeneration

When an Autoconf-generated configure script is run successfully, a shell script called **config.status** is created. This script contains all the state information generated as a result of running **configure**. If there is a need to regenerate the files created by **configure**, **config.status** can do the job quickly because it reuses the static information from when it was created without re-running all the tests. There is no global **config.status** created by **configure.pl**, but an alternate method is provided through the `--regen` argument.

When `--regen` is passed on the command line, **configure.pl** iteratively executes the individual **config.status** scripts instead of the configure scripts. This allows users to regenerate files more quickly than by running all the configure scripts again. Deciding which **config.status** scripts to run is a somewhat difficult process, however.

In a normal situation when only one **config.status** exists, it is easy for a user to decide when (and how) to run it. In the context of the VR Juggler build system, there are many configure scripts, each having its own **config.status** script. Furthermore, some **config.status** scripts may not exist based on which module (in `juggler.cfg`) was configured. As with all other cases where **configure.pl** must decide what to execute, the requested module dictates the execution behavior. Thus, if the user did not configure the default module, the `--module` argument must be passed with `--regen` to ensure proper execution of the various **config.status** scripts.

---

# Chapter 7. Extending a Module's Configure Script

To extend a module's configure script, an understanding of Autoconf is required. Without this background, it is very difficult to make good changes. Furthermore, trying to write a decent configure script from scratch is nearly impossible without first having read the basic rules of configure script structure. We will not cover those in detail since the Autoconf documentation already addresses these topics.

We will, however, explain the basic structure of (nearly) all the configure scripts in the Juggler Project. An understanding of this very simple structure makes editing the `configure.in` files much easier. We will also explain a little bit of `sh(1)` syntax and the coding conventions used in `configure.in` files.

Because there are so many `configure.in` files, consistency among all of them is very important. It only takes one script that does not follow the rules to complicate maintenance of the entire build.

## Basic Concepts

In this section, we present the basic concepts of `configure.in` files in the Juggler Project. It may be interesting to note that some of the content in this section was written in mid-1998 and has been copied verbatim. Whether this is an indication of robustness or stagnation is debatable.

## Comments

Two types of comments can be used in `configure.in`. The first is the relatively standard “#” comment. Lines beginning with this character are copied into the final configure script generated by `autoconf(1)`. The other type is the `m4(1)` comment. Lines commented in this way begin with `dnl` and are not copied into the configure script.

## Language

The scripting language used in `configure.in` is `sh(1)`. All conditional statements must use the form:

```
if test "x$some_var" = "xsome_val" ; then
    ...
fi
```

rather than:

```
if [ "x$some_var" = "xsome_val" ] ; then
    ...
fi
```

The brackets are recognized as special characters by `m4(1)` and may not be copied into `configure` properly.

When performing tests, it is recommended that some extra character be used to avoid passing any empty strings to the `test(1)` command. Some implementations do not deal well with empty strings when performing comparisons. In the above code examples, the character 'x' was used at the beginning of the two strings. While there is still old code in various configure scripts that do not use this convention, it is recommended that all new code follow this rule. Furthermore, when editing legacy code that does not protect the string operands, it is recommended that the strings be updated accordingly.

## Variable Naming

When naming variables in `configure.in`, be sure to follow the following conventions:

- Names for variables that will be set externally (primarily by command-line options such as `--enable-debug`) and not substituted into generated files at the end of execution are named in all lower-case letters. All other variables are named in all upper-case letters.
- Variables that are not substituted into generated files at the end of the configure script's execution begin with an underscore to imply that they are internal to the script. When referring to these variables, enclose the name in curly braces to clarify it fully. For example:

```
_some_internal_var='Some value'

[...]

if test [ "x${_some_internal_var}" = "xSome value" ; then
  [...]
fi
```

- Variables used to preserve an existing variable's value in case it should need to be reset are named and used as follows:

```
_vjsave_VARIABLE="$VARIABLE"

[...]

if test <Old value needs to be restored> ; then
  $VARIABLE="$${_vjsave_VARIABLE}"
fi
```

## Parameters to Macros

Parameters spanning multiple lines may be passed to Autoconf macros without using backslashes through a special block recognized by `m4(1)`. Simply enclose the parameter in `[]`'s and it will be parsed as one parameter. Some macros require the use of backslashes, however. In these cases, the argument is taken as a literal list used by `sh(1)` code. One such macro is `AC_CHECK_HEADERS( )`.

Beyond multi-line arguments, the Autoconf documentation recommends that *all* parameters passed to macros be enclosed in square brackets. This is to protect the arguments so that `m4(1)` does not mangle anything in its expansion of the macros. Consider the following examples:

```
AC_CONFIG_HEADER([vpr/vprDefines.h])
DPP_PREREQ([1.4.107])

DPP_PERL_VER([5.004], , , [AC_MSG_ERROR([*** Perl is required ***])])

AC_CHECK_PROG([MTREE_CMD], [mtree], [mtree], [$(PERL) $(scriptdir)/mtree.pl])

DPP_HAVE_GNU_MAKE([3.78], ,
  [AC_MSG_ERROR([*** The build system requires GNU make 3.78 or newer ***])])
DPP_BASIC_PROGS([$PLATFORM], [$OS_TYPE])
```

While this does make the code harder to read, it saves a lot of headache, especially between two different versions of Autoconf. Unfortunately, not all macro calls are as well-protected as those in the above example. (Doozer++ has complete argument protection, so errors are usually caused by misuses in Jug-

gler `configure.in` files.) It is recommended that all new code use square brackets. When editing existing code, add brackets when necessary.

## Custom Preprocessor Information

Anything that defines preprocessor macros other than those generated by the Autoconf macros must be defined in the module's `acconfig.h`. If they are not defined in this file, `autoheader(1)` will complain as it parses `configure.in` and finds these customizations. Follow the existing syntax and layout of `acconfig.h` when making additions to it. In general, additions only have to be made when a new `AC_DEFINE()`, `AC_CHECK_TYPE()`, etc. line is added to `configure.in`.

### Note

Autoconf versions since 2.50 discourage the use of `acconfig.h` and `autoheader(1)`. As of this writing, Autoconf 2.13 is still the most commonly available version, and as such, we cannot update to Autoconf 2.5x conventions. All Juggler Project configure scripts (and Doozer++) work with both versions.

## Structure of a `configure.in` File

We now move on to the basic structure of a `configure.in` file. As mentioned earlier, all configure scripts in the Juggler Project, save one, follow this basic structure. Each section is normally commented to distinguish its tasks from preceding and following sections.

### Initialization

The first short block of `configure.in` performs some very basic initialization tasks. The RCS revision number of the script is maintained here so that the user can verify that s/he has the most recent version of the actual configure script. A check for one file in the source tree is also made to perform a very general, inexact verification that things are where they are supposed to be. For example, VR Juggler's configure script will do its sanity check based on the existence of `vrj/vrjConfig.h`. Next, the header file created when done with the configuration process is named. In VR Juggler, this file is called `vrj/vrjDefines.h`. Finally, Doozer++ initialization occurs using the `DPP_INIT()` macro.

### Custom Arguments

In this block, the custom arguments for the configure script are defined. The first set defines the `--with-package[=arg]` and `--without-package` arguments. These are intended to be used for choosing some external software package to use at compile time. They are also used for providing a small level of customization to how the external software is found and/or used.

The second set defines the `--enable-feature[=arg]` and `--disable-feature` arguments that allow the user to choose which optional features can be built. From the Autoconf 2.12 documentation, "These options should never make a feature behave differently or cause one feature to replace another."

For both sets of options, the `AC_ARG_WITH()` and `AC_ARG_ENABLE()` macros behave the same way. The first argument is the package/feature. The second argument is the description printed when the user runs configure with the `--help` option. The third argument sets a user-defined variable in the script to the contents of `$withval` or `$enableval` respectively. If no argument is given (e.g., the user specified `--enable-opengl`), the variable will get either "yes" or "no" as its value depending upon whether the user used the positive or negative form of the argument.

After the custom arguments are defined, some of the values that may be read are processed. In particular, the application binary interface (ABI) under which the module will be compiled is defined. This is done prior to the system-dependent section to reduce duplicated code even though the ABI is highly de-

pendent upon the operating system and underlying computer architecture.

## System-Dependent Setup

Here, a common set of script variables are given values that are usually dependent upon the system on which the script is being run. Except in VPR's configure script, there is little system-dependent code. Most of it is hidden by the Doozer++ `DPP_SYSTEM_SETUP()` macro. In the case of VPR's `configure.in`, however, a large conditional block compares the value in `$target_os` (obtained from the `AC_CANONICAL_TARGET()` macro) with known systems and sets values for the variables appropriately.

Some existing variables may be modified here as well. These often include `$CFLAGS`, `$CXXFLAGS` and `$INCLUDES`. Each of these could have been set as part of the user's environment variables or previously in the script as part of executing a macro. Thus, they should be treated carefully so as not to destroy any existing values that may be stored in them.

Besides setting values in variables, custom pre-processor macros are often defined here using the `AC_DEFINE()` macro. This macro takes two parameters:

1. The name of the macro to define.
2. The value to which the macro will be set by the pre-processor.

To get something similar to:

```
#define VJ_DEF
```

use the following syntax:

```
AC_DEFINE([VJ_DEV],)
```

Defining macros that take parameters cannot be done with `AC_DEFINE()`. However, based on a check made for something (such as `sginap(2)`), such a macro can be defined at the bottom of `acconfig.h` or in the module's `Config.h` (`vprConfig.h`, `tweekConfig.h`, etc.) header after it includes the generated header.

## External Program Checks

This block is for standard program checks that configure does to ensure that software needed for compiling the package works. Using the values in several preset variables (including `$CC` for the C compiler and `$CXX` for the C++ compiler), it runs tests on the software and reports any problems it finds.

## External Library Checks

Checks for required external libraries are made here. In a configure script, a great deal of processing is often done in order to ensure that the source can be properly compiled and linked using certain external libraries. The basic process of checking for a library (say, the OpenGL library) involves first checking to see if the library can be linked. If the library can be linked, a check is then made for a standard header file associated with the library (in this case, `GL/gl.h`). In some cases, if a library cannot be found, a fallback check is provided. For example, in the case of the POSIX threads library, an attempt to link against `libpthread` is made. If that fails (as it will on HP-UX 10.20), a check is made to link against `libcma`.

Upon successfully finding a library and a needed header file, several variables are usually set that are used later in the makefile generation phase. If a library is not found, the script may simply issue a warning about the problem or it may exit altogether if the library's existence is crucial to the compiling process.

Adding new library checks can be complicated. The best method is to refer to the Autoconf documentation [<http://www.gnu.org/manual/autoconf/index.html>] and the existing checks. All follow basically the same structure.

## Header File Checks

Checks for standard header files and user-specified header files are made here. The user-specified list is a space-separated list of header files that may not exist on all systems. If a header file is found, `HAVE_FILENAME_H` is defined in the module's definition header (`vpDefines.h`, `tweek-Defines.h`, etc.) when it is generated.

## Library Function Checks

Here, checks are made for any library and system functions (e.g., `gettimeofday(3)`, `gethostbyname(2)`, `usleep(2)`) that may not exist on all systems. As with other sections, the user-specified list of functions to check is the space-separated parameter passed to the `AC_CHECK_FUNCS( )` macro. If found, `HAVE_FUNCNAME` will be defined in the generated definition header.

## Makefile Substitution

Within this block of `configure.in`, special symbols in the `Makefile.in` template files are slated for replacement with variable values. This is done through the `AC_SUBST( )` macro, and it simply replaces a tag in a `Makefile.in` with the value stored in the corresponding configure script variable. The following example illustrates what happens during the file generation phase (discussed next). Say that some `Makefile.in` contains the following line:

```
CUSTOM_VAR = @CUSTOM_VAR@
```

The string enclosed by @'s is the tag that will be replaced when `Makefile` is generated. In `configure.in`, there following line must be present:

```
AC_SUBST(CUSTOM_VAR)
```

Note that the variable is not prepended by a ``$'` because variable interpolation should not be done here. Of course, `$CUSTOM_VAR` will generally have had some value set prior to this point. That is not required, however.

It is in this phase of configuration that most of the system-dependent build work is done. This is similar to what `Imake` template files do. Virtually everything preceding this point set variables to have values appropriate to the host machine, and now they are put into the makefiles to be used for compiling the code. There are several pre-defined variables that **configure** will substitute in the input template files. Refer to the section titled "Preset Output Variables" in the Autoconf 2.13 documentation for a list of these.

## File Generation

In this last block, file generation is performed using the `AC_OUTPUT( )` macro. It iterates through its space-separated list of files and finds the associated `.in` template file. In general, this step is only used for generating makefiles (the module's definition header is added automatically due to the

`AC_CONFIG_HEADER()` call at the top of `configure.in`). When a new directory and corresponding `Makefile.in` are added to the source tree, an entry for it must be added to this list.

## Specific Modules

### Tweek

Tweek is an interesting case because it contains two distinct yet interoperable components: a C++ API and a Java API. It is possible to use one without the other, and it is even possible to configure Tweek so that neither is compiled. Of course, the latter is a rather odd case that usually only occurs when something went wrong during the execution of the configure script. Furthermore, Tweek can be compiled against any Java or C++ CORBA 2.3 (or newer) implementation. This is the result of using high-level CORBA standards and interfaces that allow ORBs swapped and allow ORBs from different vendors to communicate with each other.

---

# Chapter 8. Extending a Module's Makefiles

In this chapter, we explain how to extend a given module's makefiles. In most cases, all the modules in the Juggler Project use the same basic structure for their makefiles. Those that do not follow the existing conventions should be updated to ease maintenance issues. This chapter begins with a discussion of the basic ideas used in writing makefiles for a Juggler module. We then move on to the top-level glue makefile. Next, we discuss the individual component makefiles that typically perform the actual compiling steps. This chapter concludes with an examination of some specific modules to understand how different special cases are handled.

## Makefile Conventions

Due to the size of the Juggler Project code base, following conventions in all makefiles is very important. The uniformity makes it easier for anyone to maintain the various module build systems. If a given module does not follow the conventions, someone who does not have direct knowledge of that makefile may find it hard to fix problems that may occur. Furthermore, extensions to a given module's build system are more difficult if the same basic ideas cannot be shared across all modules. The following subsections cover most of the conventions currently in use.

### File Names

Due to Autoconf rules, all files that are to be generated by a configure script must have the suffix `.in`. Because of that, this is not so much a convention as it is a requirement. However, there are other files used in the build system, and their names are important.

Makefile stubs that make variable assignments have the string “defs” somewhere in their name. In general, such makefile stubs do not define any targets, though this may not always be the case. Examples are `make.defs.mk`, `common.defs.mk` (a Doozer++ file), and `test.defs.mk`.

In some cases, one makefile may include targets from another. The idea here is that the high-level tasks are defined in `Makefile.in`, and all the grunt work is done in the included file(s). Providing this separation can help casual viewers to understand the high-level tasks without worrying about all the behind-the-scenes details. The included files have the string “inc” in their name. The most common example seen in modules is `Makefile.inc.in`, though there could be files such as `Makefile.inc0.in`, `Makefile.inc1.in`, and so on. The use of numbering allows multiple files to be used for separating the tasks.

It is important to note that all of the makefiles named in the above examples begin with a capital letter. While it is often useful to include multiple makefiles in a single directory (of the Autoconf `.in` template variety and otherwise), there must never be a situation where a directory could contain the files `Makefile` and `makefile`. Win32 file systems cannot distinguish between the two, and only bad things can occur as a result.

### Variable Assignments

Recently, there has been a shift in the style used to perform variable assignments. The new style is based on the formatting used in all FreeBSD system and ports makefiles. This style was adopted to make formatting makefiles easier and cleaner. It is shown below:

```
VAR=          value
OPT_VAR?=    default-setting
LIST_VAR=    item1 \
```

```

        item2 \
        item3

LIST_VAR+= item4

```

The key is the whitespace around the assignment operators. On the left-hand side, there is no whitespace; on the right-hand side, there are one or more tabs. When using multiple lines, tabs are used for indentation. Spaces are *never* used. Because tabs must be used in makefile targets, it is convenient (with most editors) to use tabs for all formatting characters.

This convention is relatively new, and its adoption is not yet complete. Files that have not been updated should be reformatted when they are encountered.

## Variable References

Most implementations of **make**(1) allow the use of two variable reference styles. These are shown below:

```

VAR= value

target:
    echo ${VAR}
    echo $(VAR)

```

In the first line of the target called 'target', curly braces are used. This is the same syntax used to clarify variable interpolations in most shell scripting languages. The second line uses parentheses.

In the Juggler Project, we prefer the use of parentheses over curly braces for the following reasons:

1. It distinguishes make variable references from shell variable references. Since shell scripting can be used to a limited degree in makefiles, this can be important.
2. Some implementations of **make**(1) only support this style. While the majority of the build system is based on GNU make, some makefiles are designed to work with any implementation. Having some makefiles that use one style and others that make use of a different style can be confusing. By choosing the style that is supported by all implementations, we can never go wrong.
3. The look of the parentheses is cleaner than the curly braces. This, of course, is purely aesthetic, so it does not make the best case during style discussions (read, "arguments").

## The Glue Makefile

Each module in the Juggler Project has a *glue makefile* in its top-level directory that directs compilation of the module's main code base<sup>1</sup>. Every module's top-level makefile employs recursive calls to GNU make to build its code. The recursion occurs based on the source code directory structure. An iterative loop over each subdirectory is used make the recursive call to GNU make. Behind the scenes, all of this occurs within Doozer++ makefile stubs, but it is directed by variable settings in the glue makefile.

In this section, we will discuss how the glue makefiles work so that they may be extended correctly. We begin by explaining the basic structure employed by most of the glue makefiles found in the Juggler Project source tree. Learning how to extend those makefiles follows from that.

---

<sup>1</sup>Source code used for testing purposes of for sample applications is handled independently of the main code base, usually by a single makefile.

## Basic Structure

In all but one module in the Juggler Project, the basic structure of the glue makefile is the same: one makefile that defines high-level targets, and one included makefile that does all the work. In this subsection, we explain the way these two files work to make up the glue that holds a module's build system together. We begin with the simpler of the two: `Makefile.in`.

### Makefile.in

All modules have to have a `Makefile.in` in their top-level source directory. The real makefile is generated as part of executing the configure script. In most cases, the contents of this file is roughly the same. The file is divided into sections, and we will examine each section in turn.

### Define Default Target

As with most makefiles, the first thing done is the definition of a default target, and the glue makefile is no exception. In most cases, the default target is one that is useful to developers. In Doozer++ terms, that target is 'debug'.

Definition of a default target unfortunately has special significance if `dpp.subdir.mk` is used. In order for a recursive build to execute based on its definition in Doozer++, the file `dpp.subdir.mk` must have the special target called 'default'. Because it is not possible with GNU make to test if a target has already been defined, a workaround must be used. Namely, the variable `$(DEFAULT_SET)` must be defined if a makefile that includes `dpp.subdir.mk` defines its own 'default' target. Thus, most top-level `Makefile.in` files have the following as the first three lines:

```
default: debug
```

```
DEFAULT_SET= 1
```

It is possible that some refactoring of `dpp.subdir.mk` might eliminate the need for this. As of this writing, it is not considered a critical change.

### Include Project-Wide Variable Settings

Since VR Juggler 1.0, the build system has collected project-wide variable settings into a single file so that it may be included by all makefiles in the module's core build system. In VR Juggler 2.0, the (Autoconf-generated) file is called `make.defs.mk`<sup>2</sup>. It is always included using the following line:

```
include @topdir@/make.defs.mk
```

### Set Local Variables

Once `make.defs.mk` has been included, local variable settings are performed. These can provide further customizations on variables set in `make.defs.mk`, but usually, the variables are used only within the scope of the glue makefile. That is, the variables are needed exclusively for directing the process of building the module source code.

For example, a decision to build a certain subset of the code base may be made at this level. Alternatively, a decision about how to build the code may be made. In most cases, such a decision is whether to build profiled versions of the library (or libraries).

The most important local variable setting is that of the list of subdirectories in the module. This list is provided through the `$(SUBDIR)` variable. Due to the structure of the various modules in the Juggler

---

<sup>2</sup>A better name for this file might be `proj.defs.mk` since its job is to provide project-specific customizations in addition to the common settings provided through the Doozer++ file `common.defs.mk`.

Project, there would be nothing to do without providing Doozer++ with the list of subdirectories. The way `$(SUBDIR)` is assigned its value may vary depending on the specific structure of a given module.

## Define High-Level Targets

The main purpose of a project's `Makefile.in` is the definition of high-level build targets. These are the targets that users and developers will run to compile the module. Such targets include 'world', 'release', 'debug', and 'clobber'. The first two must be defined manually; the third and fourth are inherited through the use of Doozer++.

The high-level targets make use of low-level targets that exist in Doozer++ and, usually, in `Makefile.inc.in`. The idea is that a casual (or experienced) user can look at `Makefile.in` alone and understand how to perform common builds of a given module. `Makefile.inc.in`, with all its ugly details, is hidden—more or less.

## Include External Makefiles

Finally, the top-level makefile must include the external that usually have all the hard bits. In most cases, this means having the following two lines as the last of `Makefile.in`:

```
include $(MKPATH)/dpp.subdir.mk
include Makefile.inc
```

## Makefile.inc.in

In the context of the glue makefile, this particular file is often quite complex. If there is any one place in a module's build system that is hard to understand, it is within this file. However, the whole purpose of having this file is to collect all the complex build details into one place that is “out of the way” in some sense.

Due to the complexity and the wide variation in requirements between modules, we cannot examine a specific `Makefile.inc.in` and still provide a general explanation. Thus, we will review only the commonalities. When possible, we will cover important aspects that may not necessarily be used by all glue makefiles. Before we do that, we will begin our discussion with an review of the concepts behind `Makefile.inc.in` and why it is important.

## Basic Concepts

As has been stated previously, the fundamental idea behind the top-level `Makefile.inc.in` is to hide complexity. More explicitly, this file should contain targets that the user will not execute in most circumstances. By putting such targets in this file, a casual reader need not ever see the complexity of the glue makefile. He or she should only have to open `Makefile.in` (or the generated `Makefile`) to get an understanding of how the build works. Of course, some details are encapsulated within Doozer++ makefile stubs, so getting a complete understanding is not necessarily a straightforward process. The extensive comments at the top of and throughout `Makefile.in` (and documents such as this) are there to fill in the gaps.

While it is very difficult to separate all targets cleanly based on common usage, there are some targets that definitely fit into this file well. Such targets are only called by Doozer++ through a mechanism similar to a C callback. Users of the build system should rarely, if ever, need to execute such targets, especially because Doozer++ may set up a special execution environment for the targets. Callback targets may include the following:

- The targets used to compile the final binaries for the module's libraries, including static, dynamic, and profiled versions.
- Any targets used as “pre” and “post” steps of larger targets. Larger targets would include those used

for building and installing a module.

## Library Name Construction

The responsibility for putting all the pieces together for the final library names rests on this file. Based on variable definitions in `make.defs.mk` (see below), the final library names must be defined so that Doozer++ knows what to build. The configure script provides the platform-specific details such as file extensions so that the code in `Makefile.inc` is relatively simple. For example, assuming the existence of a variable `$(MY_LIBRARY)`, the following provides all the information needed to direct the compilation of profiled and non-profiled libraries, of both the static and the dynamic variety:

```

STATICLIB_EXT= @STATICLIB_EXT@
DYNAMICLIB_EXT= @DYNAMICLIB_EXT@

LIBS=          $(MY_LIBRARY)
STATIC_LIBS=   $(LIBS)
DYNAMIC_LIBS= $(LIBS)
    
```

Using the above variables (and more settings from `make.defs.mk`), we can construct the names of various library versions. These names are then used internally for the targets that build the actual libraries. The name construction is performed as follows:

```

MY_LIB_STATIC=      $(MY_LIBRARY).$(STATICLIB_EXT)
MY_LIB_DYNAMIC=    $(MY_LIBRARY).$(DYNAMICLIB_EXT)
MY_PROF_LIB_STATIC= $(MY_LIBRARY)$(PROFLIB_EXT).$(STATICLIB_EXT)
MY_PROF_LIB_DYNAMIC= $(MY_LIBRARY)$(PROFLIB_EXT).$(DYNAMICLIB_EXT)
    
```

### Note

The above hinges on the fact that `$(PROFLIB_EXT)` is non-empty. Otherwise, the targets we will examine in the next section would not be named correctly. If profiled libraries are not supported by the rest of the module's build system, do not use the profiled variants shown above.

## Library Build Targets

Once we know the names of the libraries we will build, we can define the targets that actually build them. In nearly every case, these targets are very close to being identical between modules. Only rarely are there module-specific variations. The following shows targets for building static profiled and non-profiled versions of `$(MY_LIBRARY)`:

```

$(LIBDIR)/$(MY_LIB_STATIC) $(LIBDIR)/$(MY_PROF_LIB_STATIC): $(OBJDIR)/*.$(OBJ
1  @$(SHELL) $(MKINSTALLDIRS) $(LIBDIR)
2  $(AR) $(AR_NAME_FLAG)$@ $(OBJDIR)/*.$(OBJEXT)
3  $(RANLIB) $@
4  cd $(LIBDIR_BASE) && $(RM_LN) $(notdir $@) && $(LN_S) $@ ./
    
```

This defines the targets for the profiled and non-profiled static libraries. Both depend on all the object files in `$(OBJDIR)`.

This ensures that the directory where the library binary will be build exists. The variable

`$(MKINSTALLDIRS)` is set via `make.defs.mk`.

These lines together build the static library. The use of `$(RANLIB)` is optional as it may not be

supported or necessary with all compilers. The configure script makes this decision and sets \$(RANLIB) accordingly.

**4** This makes a symlink to the newly compiled static library in \$(LIBDIR\_BASE). Based on variable assignments and build targets, the directory named by \$(LIBDIR) is always a subdirectory of \$(LIBDIR\_BASE). It varies based on library type (optimized, profiled, or debugging) and other platform-specific settings.

## Note

Most modules in the Juggler Project define their static library target slightly differently than the above. As of this writing, static libraries are not built on Win32 platforms because there is no easy way for the build system to distinguish between a .LIB file associated with a .DLL file and a static .LIB file. To deal with this, these projects use the following variation on the above:

```
$(LIBDIR)/$(MY_LIB_STATIC) $(LIBDIR)/$(MY_PROF_LIB_STATIC): $(OBJDIR)/*.$(OBJEXT)
ifneq (@OS_TYPE@, Win32)
    @$$(SHELL) $(MKINSTALLDIRS) $(LIBDIR)
    $(AR) $(AR_NAME_FLAG)$@ $(OBJDIR)/*.$(OBJEXT)
    $(RANLIB) $@
    cd $(LIBDIR_BASE) && $(RM_LN) $(notdir $@) && $(LN_S) $@ ./
endif
```

Next, we show similar targets used to build the dynamic versions:

```
DYLIB_DEPS= @DYLIB_DEPS@
$(LIBDIR)/$(MY_PROF_DEPS)= @DYLIB_PROF_DEPS@
$(LIBDIR)/$(MY_LIB_DYNAMIC): $(OBJDIR)/*.$(OBJEXT)
    @$$(SHELL) $(MKINSTALLDIRS) $(LIBDIR)
    $(LD) $(LDOPTS) $(DYLIB_NAME_FLAG) $(OBJDIR)/*.$(OBJEXT) $(DYLIB_DEPS)
    cp $(OBJDIR)/*.$(OBJEXT) $(LIBDIR)
ifneq (@OS_TYPE@, Win32)
    cd $(LIBDIR_BASE) && cp $(LIBDIR)/* .
endif
$(LIBDIR)/$(MY_LIB_DYNAMIC): $(OBJDIR)/*.$(OBJEXT)
    @$$(SHELL) $(MKINSTALLDIRS) $(LIBDIR)
    $(LD) $(LDOPTS) $(DYLIB_NAME_FLAG) $(OBJDIR)/*.$(OBJEXT) $(DYLIB_DEPS)
    cp $(OBJDIR)/*.$(OBJEXT) $(LIBDIR)
ifneq (@OS_TYPE@, Win32)
    cd $(LIBDIR_BASE) && cp $(LIBDIR)/* .
endif
$(LIBDIR)/$(MY_PROF_DYNAMIC): $(OBJDIR)/*.$(OBJEXT)
    @$$(SHELL) $(MKINSTALLDIRS) $(LIBDIR)
    $(LD) $(LDOPTS) $(DYLIB_NAME_FLAG) $(OBJDIR)/*.$(OBJEXT) $(DYLIB_PROF_DEPS)
    cp $(OBJDIR)/*.$(OBJEXT) $(LIBDIR)
ifneq (@OS_TYPE@, Win32)
    cd $(LIBDIR_BASE) && cp $(LIBDIR)/* .
endif
```

**1** First, we define variables to hold any dependencies the dynamic libraries have. The first is for the un-profiled dynamic library, and the second is for the profiled dynamic library.

**2** This line uses the link command (defined by the configure script) to create the dynamic library.

**3** Note that \$(DYLIB\_DEPS) is at the end of the line so that dependencies are resolved correctly.

**4** These lines deal with making copies of or symlinks to the dynamic library binary in \$(LIBDIR\_BASE). This is similar to the operation performed for the static library, shown above.

In this case, we handle the Win32 versus UNIX cases separately. The `.LIB` file generated automatically with the `.DLL` (if the library exports symbols) must be present with the `.DLL` file. On UNIX, it is sufficient to make a link only to the dynamic library.



This line again uses the linker to make the profiled dynamic library. Note that it uses `-Wl,-DYNAMIC_PROF_DEPS` to resolve dependencies. Again, we make copies of or symlinks to the profiled dynamic library.

**Note**

The separation of the targets for profiled and non-profiled versions reflects the lack of good support for profiled libraries. The support for building profiled libraries in the Juggler Project was added in late 2001, and it has not reached a level of solid maturity yet. With time, it is hoped that the the above two targets can be merged into one or that there can be better information sharing between the two.

## Installation Extension Targets

Most modules in the Juggler Project install more than just libraries and header files. Data files, sample programs, and other useful bits may be installed as well. Even before something can be installed, an installation directory hierarchy usually needs to be in place. Doozer++ offers features for the basic installation and for extending the installation. In this section, we explain how and why modules in the Juggler Project extend the basic Doozer++ installation capabilities.

First, all modules are required to ensure that the installation hierarchy they need exists before installing. This is accomplished by telling Doozer++ that a 'beforeinstall' target exists as follows:

```
BEFOREINSTALL= beforeinstall

beforeinstall:
    <create installation hierarchy>
```

If the variable `$(BEFOREINSTALL)` is defined, the Doozer++ installation targets will invoke the target(s) defined in `$(BEFOREINSTALL)` before installing anything (hence the name). (Using this technique is necessary because GNU make provides no way for makefiles to determine if a target has been defined.) Most modules make use of `mtree(1)` (or the work-alike Perl script) to create the installation hierarchy in a target called 'hier'. More information about the `mtree(1)` command is available in Appendix A, *Helper Scripts*.

Next, modules may need to take extra steps before installing specific builds of the libraries (such as those built by the targets 'dbg', 'prof-dso', etc.). This need is indicated as follows:

```
PREINSTALL= pre-install

pre-install:
    <perform steps immediately prior to library installation>
```

If GNU make were more expressive, the above would probably be done using targets such as 'pre-dbg-install', 'pre-prof-dso-install', and so on.

The complement of pre-installation is post-installation. Availability of custom post-installation targets is specified using the `$(POSTINSTALL)` variable. However, in Doozer++ 1.5, there is some inconsistency with the use of post-installation targets. They do not correspond exactly to the pre-installation targets, and for that reason, some refactoring is needed. In Doozer++ 1.5, the post-installation targets are executed after a full installation is done rather than after each library version is installed. For that reason, the current `$(POSTINSTALL)` variable should probably be renamed to `$(POSTLIBSINSTALL)` or some variant thereof. The current post-installation target always installs the module's header files and then runs any custom post-installation steps. It is at this point that most modules install their data files, sample programs, and other necessary/helpful files.

Finally, the complement of `$(BEFOREINSTALL)` is executed. It is specified using the variable `$(AFTERINSTALL)`. Considering that most modules use `$(POSTINSTALL)` for data file installation, there is not much left for `$(AFTERINSTALL)` to do.

## Developer Installation Targets

At long last, we have reached the point where we explain how the developer installation is created. All the modules in the Juggler Project are required to set up a developer installation, though they are free to implement it any way they want. The only restriction is that the modules must not touch or otherwise modify the developer installations of previously built modules. As was stated in the section called “The Developer Installation”, the idea here is to create something that *looks like* a full installation within the developer's build tree. No special steps should be taken that would cause the developer installation to look or behave any differently than a full installation.

As explained in Chapter 4, *The “Global” Build*, all modules must define targets that set up and tear down the developer installation. The set-up target is always called 'links', and it can be implemented exactly as follows:

```

1 instlinks= $(topdir)/instlinks
  AFTERBUILD= afterbuild

  afterbuild:
5      @$(MAKE) links

  links:
  ifdef BUILD_TYPE
      $(MAKE) links-$(BUILD_TYPE)
10 else
      $(MAKE) links-all
  endif

  links-all:
15      @$(MAKE) EXTRA_INSTALL_ARGS=-l prefix="$(instlinks)" installworld

  links-dbg:
      @$(MAKE) EXTRA_INSTALL_ARGS=-l prefix="$(instlinks)" install-debug

20 links-opt:
      @$(MAKE) EXTRA_INSTALL_ARGS=-l prefix="$(instlinks)" install-optim

  links-prof:
      @$(MAKE) EXTRA_INSTALL_ARGS=-l prefix="$(instlinks)" install-profiled
25

  clean-links:
  ifndef GLOBAL_BUILD
      rm -rf $(instlinks)
  endif

```

The above makes use of a key feature Doozer++ 1.5: the `$(BUILD_TYPE)` variable. When the target 'afterbuild' is invoked, Doozer++ tells the target what type of build was just performed using this variable. Based on that, the 'links' target can execute the right installation.

### Note

The `$(instlinks)` variable (which must be used as the prefix for the developer installation) will be overridden by the global build so that all modules make their developer installations in the same directory. The setting on line 1 of the above block provides the default value. This is useful when the module is being build standalone (i.e., without using the global build).

When performing that installation, another Doozer++ 1.5 feature is used: the BSD-compatible `install(1)`

script. Prior to Doozer++ 1.5, installations were done under the assumption that **install-sh** might have to be used if a BSD-compatible **install(1)** command was not available. The **install-sh** script is only partially compatible with BSD **install(1)**, and because of limitations in that script, installations could be very slow and complex, even when the **install(1)** command was available. Doozer++ 1.5 ships with a Perl script called **bsd-install.pl** that is fully compatible with BSD **install(1)**, and it adds a handy extra feature: symlink creation. When the **-l** option is passed, and the target OS supports symlinks, symlinks to the source files are created instead of making copies of the files. This is exactly what is needed for developer installations, and the above **'links-\*** targets make use of that option through the Doozer++-recognized variable `$(EXTRA_INSTALL_ARGS)`.

## Include Build and Clean-Up Functionality from Doozer++

The last thing that must be done by `Makefile.inc.in` is fairly straight forward. The Doozer++ functionality for building everything and for cleaning up must be imported. Most modules make use of the following code as their last few lines:

```
_clobber:
    @$(MAKE) cleandepend
    @$(MAKE) clean-links

_LOCAL_CLOBBER= 1

include $(MKPATH)/dpp.libs.mk
include $(MKPATH)/dpp.clean.mk

CLEAN_DIRS+= $(BUILDDIR_BASE) $(LIBDIR_NAME)
CLOBBER_DIRS+= $(BUILDDIR_BASE) $(LIBDIR_NAME)
```



1 This indicates that a local 'clobber' target is available to be executed after the default Doozer++ 'clobber' target.  
 2 This includes the Doozer++ file `dpp.libs.mk` which defines all the functionality described in the preceding sections.  
 3 These lines include the basic Doozer++ clean-up code and extend the list of directories removed by the 'clean' and 'clobber' targets respectively.

## Auxiliary Files

We now turn our attention to auxiliary files used by the glue makefile and the component makefiles (discussed in the section called "Individual Component Makefiles"). Most modules have only two auxiliary makefiles in their top-level directory: `common.defs.mk.in` and `make.defs.mk.in`. We discuss only those two files here; any extra files are project-specific and could probably be merged into `make.defs.mk.in` or `Makefile.inc.in` somehow.

### `common.defs.mk.in`

This is a file that comes with Doozer++ (in the `examples` directory). It provides a (nearly) comprehensive list of all the variables that may be defined by Autoconf macros in Doozer++. The idea is that a project developer can make use of this file with Doozer++ macros so that he or she does not have to know all the variables that may be substituted by the macros. In addition to the variable settings, there is some code at the bottom of this file for handling different ABIs, though at some point, this may move into a formal Doozer++ makefile stub.

### `make.defs.mk.in`

This file performs three simple tasks:

1. Include `common.defs.mk`

2. Provide project-specific customizations that extend settings in `common.defs.mk`
3. Provide the basic name of the library (or libraries) that will be compiled

In reality, these two steps are usually reversed in the actual file, but it makes more sense to discuss them using the above ordering.

### **Include `common.defs.mk`**

Including `common.defs.mk` is a relatively straightforward operation. As of this writing, all modules have the following as the last line of their `make.defs.mk.in` file:

```
include $(topdir)/common.defs.mk
```

The variable `$(topdir)` is available through an assignment made earlier in the file:

```
topdir= @topdir@
```

The use of `$(topdir)`, which always has an absolute path, is required so that this file can be included by any other makefile in the module's source tree.

### **Provide Project-Specific Customizations**

Project-specific customizations are made using the extension mechanisms build into variable assignments in `common.defs.mk`. In all cases, a variable whose name begins with `EXTRA_` is used to extend another variable's default settings. For example, `$(CXXFLAGS)` is extended using `$(EXTRA_CXXFLAGS)`. Extensions can be performed using the `+=` operator, though such operations should be performed after `common.defs.mk` is included.

### **Name the Library (or Libraries)**

This is actually part of the project-specific customizations, but it is important to distinguish it so that it is not overlooked. The basic idea here is that one or more variables, one for each C/C++ library, will be defined. These variables will contain the basic name of the library without any extensions or version information. Furthermore, the library name will vary correctly based on the target platform. On UNIX-based systems, that means that the library will have the prefix "lib", and on Win32 platforms, there will be no prefix. For example:

```
TWEEK_LIBRARY= @LIB_PREFIX@tweek
```

The library name variables will be used by `Makefile.inc.in` when constructing the list of libraries that Doozer++ must build.

## **Extension**

Unless there are dramatic changes to the way Doozer++ does its job at the time of compilation, the file `Makefile.in` will remain fairly static. Extensions should almost always go in `Makefile.inc.in` or `make.defs.mk.in`, depending on the nature of the extensions. User changes should *never* be made to `common.defs.mk.in`. This file provides a reasonably complete set of variables for extending its behavior, and `make.defs.mk.in` serves as an ideal place to perform overrides to any settings made in `common.defs.mk.in`. The real problem occurs when it is necessary to copy a modified version of `common.defs.mk.in` from Doozer++. Merging local changes can be hard unless `common.defs.mk.in` is always imported onto a CVS vendor branch.

## Individual Component Makefiles

The recursion performed by the glue makefile would have nothing to do were it not for the component makefiles in each subdirectory. The job of these files is to provide the list of source files that must be compiled. When the recursion comes to a given subdirectory, a target ('dbg', 'opt', etc.) is executed. Based on that target, the compiler argument list is constructed, and the source files are compiled iteratively (or in parallel depending on arguments given to **make**(1)).

In most cases, the targets are defined in Doozer++ makefile stubs such as `dpp.obj.mk`. It is up to the build system author to ensure that the right makefile stubs are used together. Doozer++ 1.5 provides only a few such stubs, and they all cooperate well. For example, `dpp.libs.mk`, `dpp.libs-basic.mk`, and `dpp.simple-app.mk` all work seamlessly with `dpp.obj.mk`, `dpp.subdir.mk`, and `dpp.obj-subdir.mk`.

## Basic Structure

In the following sub-subsections, we examine the basic structure of a component makefile. This is almost a line-by-line examination of a given makefile. We concentrate our discussion on the compilation of C/C++ code since that makes up the majority of the Juggler Project. Java and IDL are handled somewhat differently, though the basic ideas are the same. The major difference is that no dependency files are generated for Java or IDL using Doozer++ 1.5.

## Default Target

The default target should be defined before doing anything else. This prevents any external files that are included from inadvertently defining a target that would be implicitly defined as the default target. In most cases, the default target will be 'all' or 'debug'.

## Include `make.defs.mk`

The next step should be to include `make.defs.mk` to get all of its variable definitions. Some of these may be needed later in the execution of the including file, so it should be included as early as possible. It should always be included using the following syntax for the path:

```
include @topdir@/make.defs.mk
```

This will ensure that the path is specified correctly when the actual makefile is generated. Once `make.defs.mk` is included, `$(MKPATH)` can be used for including files in the `Doozer++/mk` directory of the source tree.

## Preset and “Local” Variables

The first set of assignments is for preset variables (that is, those that are substituted by **configure** when the makefile is generated) and for variables that are closely related to the preset variety. In general, many of these variables will have values that are specific to the current makefile. In other words, they cannot be shared via a common file that would be included by all makefiles.

Depending on which makefile is being examined, any extra “miscellaneous” variables needed are also set here. For the most part, everything set here is local to the current file.

There are several significant variables that must be set. They are as follows:

`includedir`      The directory where headers are installed, if there any headers to install. The installed files is contingent upon the `$(INSTALL_FILES)` variable (see below).

srcdir	The directory containing the source code to be compiled. Normally, this is assigned a value based on the <code>@srcdir@</code> substitution variable.
INSTALL	The path to the installer program. This must be included in every component makefile because it may be the <b>install-sh</b> script distributed with Autoconf. In that case, the path will be relative, and it will vary with each level of nesting in the source tree.

There are also some optional variables that may be necessary depending on the circumstances. They are:

SUBOBJDIR	The subdirectory of <code>\$(OBJDIR)</code> (defined by Doozer++) where the object files compiled by this component makefile go. In most cases, it is helpful to use this variable, especially when building multiple libraries.
SUBDIR	This is the list of all subdirectories of the current directory where compiling needs to be done. It is used in conjunction with <code>dpp.subdir.mk</code> or <code>dpp.obj-subdir.mk</code> to perform recursive calls to <b>make(1)</b> . These recursive calls to <b>make(1)</b> are handled wherever other subdirectories exist—not just in the top-level glue makefile.
INSTALL_FILES	The files to install. If it is not set, it defaults to <code>\$(srcdir)/*.h</code> . It may be extended (using the <code>+=</code> operator) or overridden based on specific needs. Extensions must be performed after including the Doozer++ <code>.mk</code> file(s), however. See below for more details on including these files.

## Listing Source Files

Next, we provide the list of source files that will be compiled. This is also used by **makedepend(1)** (or other dependency-generating tool) to get dependencies. For C++ code, the list of sources must be assigned to the `$(SRCS)` variable. These file names should not have any path prepended to them. When the dependencies (the so-called “.d files”) are generated, it knows which directory contains the files to parse from the `$(srcdir)` variable.

In some cases, some files may be conditionally compiled depending on the results of running the configure script. To add these source files to `$(SRCS)`, insert something similar to the following before making the list of object files:

```
ifeq (@SOME_VAR@, "YES")
    SRCS+= extra_src.cpp
endif
```

## Include All Necessary Doozer++ .mk Files

Once all the variables are defined, this is a good time to include any of the `.mk` files that this file will need. The path to these files can be extracted from the variable `$(MKPATH)` that is defined via `make.defs.mk`. Typically, these will be the files needed for compiling, generating dependencies and installing.

## Local Targets

Some makefiles may need targets for any source files that are not compiled into the module's library but are used, for example, as test code. This is one example of a time when it may be convenient to compile code in the current directory rather than in a common object file directory. Generally, these targets will be similar to the library targets; however, dependencies will not be generated for these files. The Doozer++ dependency-generation code only looks at `$(SRCS)`. A local target for generating dependencies

could be defined if so desired.

## Clean-Up Information

After including one of the Doozer++ compiler makefile stubs, a makefile can add to the list of files and/or directories removed when the 'clean' and 'clobber' targets are run. To extend these targets, add to (+= operator) one or more of the following variables:

CLEAN_FILES	The list of files removed by the 'clean' target.
CLEAN_DIRS	The list of directories removed by the 'clean' target.
CLOBBER_FILES	The list of files removed by the 'clobber' target.
CLOBBER_DIRS	The list of directories removed by the 'clobber' target.

## Dependencies

Most `Makefile.in` files end with an area where all the source dependencies are loaded via the GNU make include mechanism. In general, it appears as follows:

```
-include $(DEPEND_FILES)
```

The variable `$(DEPEND_FILES)` is constructed by Doozer++ based on the list of C/C++ sources (see the section called "Listing Source Files" above). The use of `-include` versus `include` silences warnings when the dependency files have not been created yet.

Some makefiles wrap the line using GNU make conditional constructs. In particular, most component makefiles actually use the following line:

```
ifndef DO_CLEANDEPEND
    -include $(DEPEND_FILES)
endif
```

This is a workaround for a deficiency in GNU make. Namely, the target that is being executed cannot be tested. As a result, cleaning up dependencies with the 'cleandepend' target is not straightforward. Doozer++ defines that target, and when it is executed, it defines the variable `$(DO_CLEANDEPEND)`. If, at some point, GNU make had features to test the target being executed, this workaround (and many, many others) could be removed.

## Extension

Extension to a given component makefile is almost always done by adding more source files to the `$(SRCS)` variable assignment. Whether this is done using conditional constructs or extending the always-build list depends on the circumstances. Other extensions may come in the form of new files that may be generated from an IDL file or an extension to the list of files to be installed.

## Specific Modules

Some modules must do more work than simply compiling C++ code into a library. To provide readers with an understanding of how to handle specific cases that are not entirely uncommon, we examine how various modules in the Juggler Project deal with these situations. The cases we will examine are as follows:

- Multiple C++ libraries
- Compiling Java libraries
- Generating Java and C++ code from IDL

## **Multiple Libraries in VR Juggler**

### **Java in Tweek**

### **IDL in Tweek**

---

## **Part IV. Appendices**

---

---

## Table of Contents

A. Helper Scripts .....	53
<b>mtree(1) and mtree.pl</b> .....	53
<b>install-dir.pl</b> .....	53
<b>install-src.pl</b> .....	53
<b>makefiles-gen.pl</b> .....	54
InstallOps Perl Module .....	54
<b>make-ver.sh</b> .....	55
<b>mkmakefile.pl</b> .....	56
B. Build System Usage .....	57
C. GNU Free Documentation License .....	58
PREAMBLE .....	58
APPLICABILITY AND DEFINITIONS .....	58
VERBATIM COPYING .....	59
COPYING IN QUANTITY .....	59
MODIFICATIONS .....	60
COMBINING DOCUMENTS .....	61
COLLECTIONS OF DOCUMENTS .....	62
AGGREGATION WITH INDEPENDENT WORKS .....	62
TRANSLATION .....	62
TERMINATION .....	62
FUTURE REVISIONS OF THIS LICENSE .....	63
ADDENDUM: How to use this License for your documents .....	63

---

# Appendix A. Helper Scripts

The VR Juggler build system makes use of many, many helper scripts. Most are written in Perl, though a few are written in `sh(1)` because their task is very simple. In this appendix, we present information on how each script fits into the larger picture of the VR Juggler “global” build system.

## `mtree(1)` and `mtree.pl`

`mtree(1)` is a utility for mapping directory hierarchies. It dates back to 4.3BSD-Reno, and development of it has continued in FreeBSD and in NetBSD. `mtree.pl` is a Perl script written to mimic the behavior of `mtree(1)` (the version distributed with FreeBSD 3.0) so that Juggler's installation can take advantage of its features on platforms without `mtree(1)`. The two are not completely functionally equivalent at this time, but the basic features needed by VR Juggler are present.

The key aspect of `mtree(1)` that the Juggler installation uses is the creation of directory hierarchies from specification files (named with the extension `.dist`). These specifications can be used to create the full installation tree with all directories guaranteed to have the same permissions and ownership. The actual specification files are generated by `configure` from template files to allow maximum flexibility. One key file, `VJ.lib.dist`, is generated by `configure` from scratch (i.e., without a template) so that it can take advantage of `configure`'s internal knowledge about the target platform.

`mtree(1)` provides a wide range of features but unfortunately, not all are implemented in `mtree.pl`. For example, besides being used to build a hierarchy, it can be used to generate a specification file for an existing hierarchy. For more information about what can be done with this utility, refer to the manual page if `mtree(1)` is distributed with the operating system. More work on bringing the two closer together in functional equivalence is anticipated.

## `install-dir.pl`

This Perl 5 script is used to install an entire directory tree without any of the CVS directories. Its use is very simple. In general, the base of the tree to be installed and the destination directory are all that is necessary. The basic usage is:

```
install-dir.pl [{"-i source directory"} | {"-o destination directory"}] [{"-u user name"} | [{"-g group name"} | [{"-m mode"} | [{"-e extension list"} | [{"-l}]]
```

This script is used when an entire directory (except for CVS-related files) must be installed. No other files are skipped during the installation process. Therefore, it is useful when a data directory must be installed.

The `-e <extension list>` option shown above restricts `install-dir.pl` so that it installs only the files in the extension list. The extension list is a comma-separated list of extensions of the form `.ext`.

## `install-src.pl`

Similar to `install-dir.pl`, this script installs the contents of a directory tree to a specified location. It is, however, more selective in what it installs. Only source files of *case-insensitive* recognized types (`.txt`, `.c`, `.h`, `.cxx`, `.cpp`, `makefiles`, `Makefile.in` files, and so on) are copied from the source to the destination. Any CVS directories are not copied over. The options for this script are as follows:

```
install-src.pl [{"-i source directory"} | {"-o destination directory"}] [{"-u user name"} | [{"-g group name"} | [{"-m mode"} | [{"-e extension list"} | [{"-l}]]
```

In addition to the hard-coded list of extensions, users can provide their own extensions using `-e`

<extension list> where <extension list> is a comma-separated string of extensions of the form .ext. Of course, if the extension list gets long enough, it might be worthwhile to consider using **install-dir.pl** instead.

## Note

With time and changing requirements in individual modules, the scripts **install-src.pl** and **install-dir.pl** have grown closer and closer to each other in their functionality. While they do share code through the `InstallOps` Perl module, there is some duplication. At this point, the only real difference between the two is that **install-src.pl** has a preset list of recognized extensions. This list could be passed to **install-dir.pl**, or **install-dir.pl** could be extended to enable this preset list in addition to anything passed in with `-e`.

## makefiles-gen.pl

This script is used to perform the same functions as **configure** when it expands the “@. . .@” strings in .in template files. Its use is reserved for the installation phase rather than the configuration phase. The installation process copies over many `Makefile.in`'s that **configure** would normally generate for use by developers. To simplify maintenance, the same template files are used for installations that users can access. Since the templates still contain the “@. . .@” strings, they must be expanded again. Thus, **makefiles-gen.pl** steps in for this purpose.

Options for `makefiles-gen.pl` are slightly different from any of the other scripts. It uses GNU-style options for the sake of clarity since some of the options correspond directly to a “@. . .@” string to be expanded. The options described here are required unless otherwise noted.

The key to this script is `vars.pl.in`, located in the top-level VR Juggler source directory. It is a Perl file that can be evaluated at run time by the script to set values used for substitution. The format used in the file is:

```
$VARS{ 'VAR_NAME' } = '@VAR_NAME@' ;
```

Note that `VAR_NAME` should be the same in both places for ease of understanding. The `%VARS` hash is used in the substitution process. Occurrences of its keys found in a `Makefile.in` are replaced with the value associated with that key (the expanded “@. . .@” string). Also note the use of apostrophes to prevent expansion of variables by the Perl interpreter. Be careful to use the right string delimiters when adding new values.

## InstallOps Perl Module

Many of these Perl scripts use the `InstallOps` module (found in `InstallOps.pm` with the other scripts). This module provides subroutines implementing commonly used techniques in working with directory trees and files during installation. Ideally, it is general enough to be used in writing new scripts that can be added to the installation process. A short description of the available subroutines and their use is provided here.

```
recurseDir( );
```

```
( $start_dir, $base_inst_dir );
```

Recurse through the given directory tree beginning at `$start_dir`. The second argument, `$base_inst_dir`, names the base directory to which the files will be installed. This routine requires that the calling package define a subroutine called `recurseAction( )` that defines what actions to take when a normal file is encountered during the recursion process. It is only called when non-directory type files are found in the current directory of the recursion process. Since this is specialized for installation, it ensures that the destination directories (rooted at `$base_inst_dir`) exist.

**newDir();**

`($base_dir, $newdir);`

Create a new directory (`$newdir`) in a given directory tree (`$base_dir`). If it already exists, it is not created.

**installFile();**

`($filename, $uid, $gid, $mode, $dest_dir);`

Install a given file (`$filename`) with specified permissions (`$mode`) in UNIX octal style to a destination directory (`$dest_dir`). Ownership is set based on the provided values for `$uid` and `$gid` (which has no effect on a Win32 platform).

**replaceTags();**

(Replace tags of the form `@...@` in the given file (`$infile`) with known replacement values (found in `%VARS`). The tags are the keys of `%VARS`, and the replacement values are the values associated with those keys.);

## make-ver.sh

This is a simple script used by most of the modules to generate a header file defining version information for the module. It reads the `VERSION` file used by all modules to create this header. Its usage is as follows:

```
make-ver.sh [[-b branch]] [[-d date]] [[-f version file]] [[-i source template file]]
[-n release name]] [[-o output file]] [[-s threading subsystem]] [[-v version]]
```

The following shows a typical usage in a makefile:

```
PARAM_HEADER= proj/projParam.h
BRANCH=      HEAD
CANON_NAME=  Chef
VER_ARGS=   -f $(PROJROOT_ABS)/VERSION -s @VPR_SUBSYSTEM@ \
            -b $(BRANCH) -o $(PARAM_HEADER) -n "$(CANON_NAME)" \
            -i $(PROJROOT_ABS)/proj/projParam.h.in \
            -d "`date '+%b %e, %Y %H:%M:%S'`"

BEFOREBUILD= beforebuild

beforebuild:
    @$(MAKE) $(PARAM_HEADER)

$(PARAM_HEADER): $(PROJROOT_ABS)/proj/projParam.h.in $(PROJROOT_ABS)/VERSION
    @$(SHELL) $(scriptdir)/make-ver.sh $(VER_ARGS)
```

The above would normally occur in a module's top-level `Makefile.inc.in` file. It only generates the header `proj/projParam.h` when the source template changes or the module's `VERSION` file changes. Because it is generated during 'beforebuild', it is guaranteed to exist when any C/C++ code would need to include it for proper compiling.

This script basically serves as a way to generate the version information at build time rather than at configuration time using the configure script. Clearly, the use of a `.in` template file would be suited well to life as a **configure**-generated file, but the needs of the Juggler Project dictate that version information be up to date for builds since they are done more frequently than configurations.

## mkmakefile.pl

This is a script that can generate Doozer++-aware makefiles. It can generate two types of makefiles: module component makefiles and application makefiles. (The component makefiles of modules in the Juggler Project were discussed in the section called “Individual Component Makefiles”.) The script also makes use of information provided through m4 macros defined in `juggler/macros/vrj-helpers.m4`. For that reason, it cannot be distributed with Doozer++. In other words, this script is tailored for use within the Juggler Project build system and cannot be used outside of that scope without modification.

Use of this script is rather complicated, but it is probably easier than writing the makefiles by hand. Due to the complex nature, it is recommended that interested readers refer to the usage information printed by executing the following:

```
mkmakefile.pl --help
```

The output is quite comprehensive, and any usage information included within this document would simply be verbatim copies of that output. However, the following explanation, taken from the comments at the top of the file provide a good description of what this script can do.

This script can generate three types of template `Makefile.in`'s for use with a configure script:

1. A Doozer++ makefile that can do one of the following three build processes:
  - A. Build object files from an auto-detected list of sources coming from one or more user-specified directories
  - B. Recurse through a list of directories
  - C. A and B together
2. A makefile capable of compiling a single application from an arbitrary auto-detected list of sources from an arbitrary list of user-specified directories.
3. A makefile capable of compiling multiple applications from an arbitrary list of user-specified sources (compared against an auto-detected list for error checking) from an arbitrary list of user-specified directories.

Given the right information, a “real” `Makefile` can be generated automatically for any of the above types as part of the creation process.

The makefiles generated can be used without modification, though in typical use, they will probably serve as more of a starting point requiring only minor modifications.

---

# Appendix B. Build System Usage

Refer to the *Juggler Team Guide* for VR Juggler 2.0 (and beyond) for build system usage information.

---

# Appendix C. GNU Free Documentation License

Version 1.2, November 2002

## FSF Copyright note

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sec-

tions then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you

as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

### GNU FDL Modification Conditions

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the

title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the

combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

## COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

### Sample Invariant Sections list

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

### Sample Invariant Sections list

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

---

# Glossary

## A

- Autoconf A tool used to generate cross-platform shell scripts that test the functionality of software development tools such as compilers and of local software installations. The results of the tests can be used to configure a software build system to work with the local system setup. Autoconf-generated scripts aid in the development of cross-platform software by detecting and managing platform-specific idiosyncrasies. Autoconf is part of the GNU software distribution.
- Automake A tool used in conjunction with Autoconf to generate complex makefiles from simple input files. Automake is part of the GNU software distribution.

## C

- configure script A shell script, generated by the **autoconf** utility, that tests capabilities of common developer tools such as compilers and linkers and sets up a build environment based on the results of the tests.

## D

- dependency graph A directed, acyclic graph defining connections between components. The connections imply a dependency of the source component on the target component.

## G

- glue makefile A makefile that pulls together smaller makefiles and/or directs the compilation of a collection of makefiles in a source tree. In terms of the VR Juggler build, this usually means controlling the recursion through a source tree when building a module.

## I

- IDL compiler A software tool that reads an *IDL* file and generates code in a specific language (e.g., Java, C++, Smalltalk, etc.). This code may be anything, but typically, it is stub or skeleton code that is extended by user-defined code that completes the interface implementation.
- Interface Definition Language The Interface Definition Language is a simple language used to define the interface (or signature) an object will have. An interface is composed solely of methods (functions) that may be invoked on an object (data element). No data members are present in the interface definition. IDL is not tied to a specific language but instead

must be compiled into another language where the interface will be implemented.

## J

JAR file                      A Java archive. These files typically contain everything needed to execute a Java application or a Java applet. They may also be used to package language-independent information such as data files or images.

## M

makefile stub                A makefile that cleanly encapsulates some specific task. This may be variable assignment or definition of reusable build targets. Many makefile stubs are only a few lines long, though some may be very complex depending on the task. The idea is that the makefile can be plugged into a larger makefile to promote reuse of **make(1)** code.

## P

Plain Old Documentation     The documentation format used by Perl. It may be embedded within Perl scripts so that the documentation is bundled with the script at all times. The documentation can be rendered to three formats: man-page, HTML, and plain text. To get a script's documentation in the same style as that used by the **man(1)** command, use the following (replacing `<script-name>` appropriately):

```
perldoc <script-name>
```

## S

substitution variable        A shell variable within an Autoconf-based configure script that is substituted in a `.in` template file. For the variable `$variable`, the `Autoconf@variable@` syntax is used in the `.in` template file. This is not a formal term used in Autoconf documentation but rather a term used within the VR Juggler build system documentation to distinguish these variables from other shell variables in a configure script.

## X

XSL Transformations        One of two parts that make up the Extensible Stylesheet Language (XSL). This part defines an XML-based programming language used to perform transforms on XML documents.

---

# Index

## Symbols

- config script
  - dependency management, 17
  - design of, 16
  - generation of, 18
  - job of, 17
- .m4 file
  - design of, 18

## A

- Ant, 5
- Automake
  - problems with, 3

## B

- BASH
  - in makefiles, 8
- bsd-install.pl, 21

## C

- common.defs.mk, 37, 39, 46, 46
  - use by glue makefiles, 45
- config.status, 30
- configure.pl, 14
  - build configuration, 28
  - command-line argument handling, 27
  - configuration file, 14
  - default arguments
    - loading of, 28
  - design of, 14
  - extension of, 27
  - file regeneration, 30
  - help output, 28
  - module configure scripts
    - execution of, 29
  - reconfig
    - generation of, 29
  - top-level makefile
    - generation of, 29

## D

- developer installation, 20
  - differences from release, 20
- Doozer, 5
- Doozer++, 1, 2, 7, 7, 8, 13, 21, 32, 33, 33, 34, 37, 38, 39, 40, 40, 40, 43, 44, 45, 45, 46, 46, 47, 47, 48, 48, 49, 49, 56
  - goals of, 3
- dpp.libs-basic.mk, 47
- dpp.libs.mk, 47
- dpp.obj-subdir.mk, 47, 48
- dpp.obj.mk, 47, 47

- dpp.simple-app.mk, 47
- dpp.subdir.mk, 39, 47, 48

## G

- gadgeteer-config, 17, 17

## I

- install-dir.pl, 53
- install-src.pl, 53
- InstallOps.pm, 54
- instlinks, 16 (see developer installation)

## J

- jccl-config, 17, 17, 18
- juggler.cfg, 15, 26, 28, 30
  - basic format
  - example of,
  - with environment variables
  - example of,
- JugglerConfigure.pm
  - extension of, 26
  - JugglerConfigure package, 26
  - JugglerModule package, 26
  - ModuleDependency package, 27

## L

- languages
  - supported, 4
  - unsupported (currently), 4

## M

- make-ver.sh, 55
- makefiles-gen.pl, 54
- mkmakefile.pl, 56
- mtree, 53
- mtree.pl, 53

## R

- required targets, 10
  - build, 10
  - clean-up, 12
  - developer, 13
  - installation, 11
  - multi-step, 11

## S

- sonix-config, 17

## T

- tweek-config, 16, 17

## V

- vpr-config, 16, 18
- vrj-helpers.m4, 56
- vrjuggler-config, 17