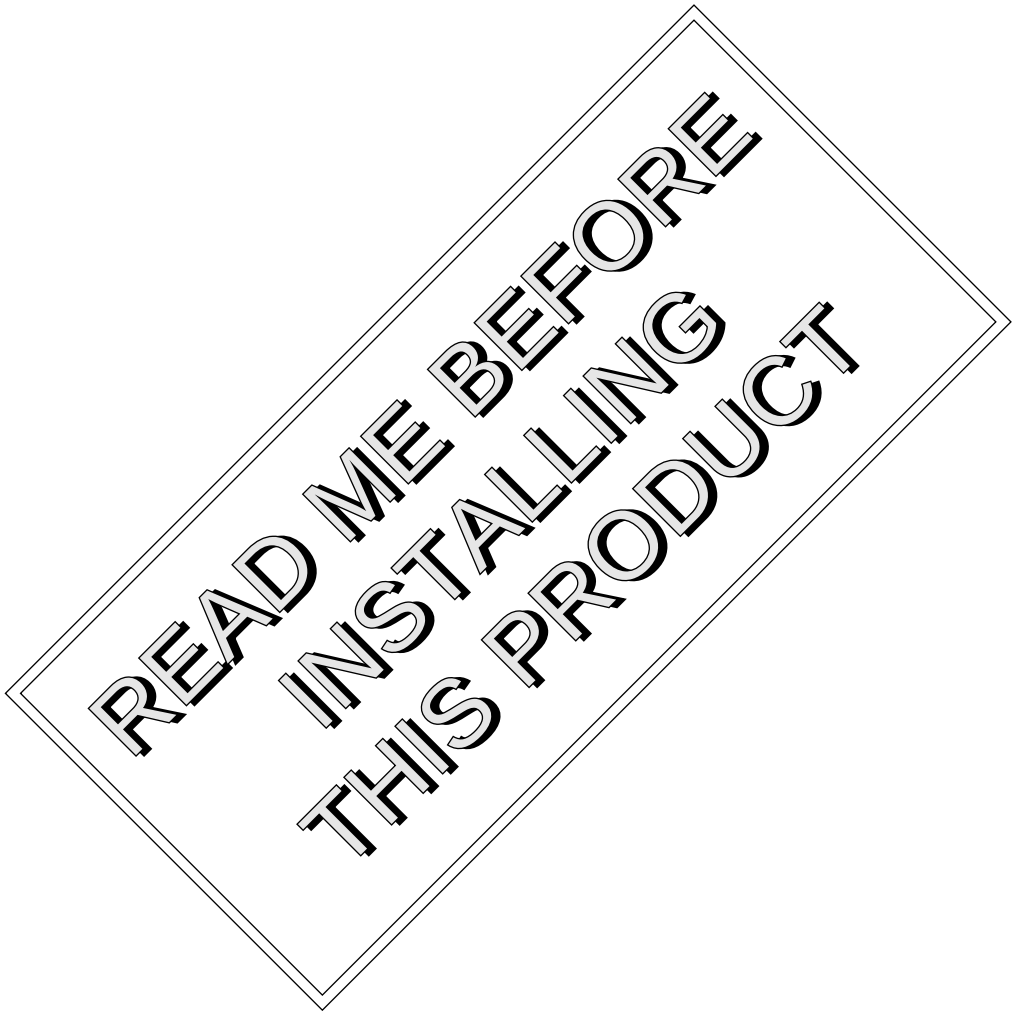
Concurrent C/C++

Version 5.2 Release Notes (Linux)

April 2001

0898497-5.2



CONCURRENT
COMPUTER
CORPORATION

Copyright

Disclaimer

The information contained in this document is subject to change without notice. Concurrent Computer Corporation has taken efforts to remove errors from this document, however, Concurrent Computer Corporation's only liability regarding errors that may still exist is to correct said errors upon their being made known to Concurrent Computer Corporation.

License

Trademark Acknowledgments

Contents

1.0. Introduction

Concurrent C/C++ is part of the PowerWorks™ Linux Development Environment (PLDE) and utilizes Edison Design Group's C++ front end and Concurrent's Common Code Generator (CCG) technology to produce highly optimized object code tailored to Concurrent systems running PowerMAX OS™.

There are several compiler switches that provide a degree of compatibility with previous drafts of the ANSI C++ Standard, USL versions 2.1 and 3.0 C++ compilers (also known as **`cfront`**), Kernighan and Ritchie C and SVR4 C, in addition to compatibility with the ANSI C++ and C languages.

This release combines into a single compiler what were, previous to release 5.1, separate C and C++ compilers. Release 5.1 was the "next" release after Concurrent C Compiler 4.3 and Concurrent C++ Compiler 3.1. This release does not require un-installing previous versions of the C and C++ compilers. It installs in its own unique location and provides a mechanism for supporting the use of multiple releases.

This release provides a command line-based program development environment for building complex projects. Alternatively, the user may use NightBench™ to interface with this program development environment.

As of release 5.1, C/C++ no longer ships with USL iostream and complex libraries. They are available separately.

2.0. Documentation

Table 2-1 lists the Concurrent C/C++ 5.2 documentation available from Concurrent.

Table 2-1. Concurrent C/C++ Version 5.2 Documentation

Manual Name	Pub. Number
Concurrent C/C++ Reference Manual	0890497-020
Concurrent C/C++ Version 5.2 Release Notes (Linux)	0898497-5.2

Copies of the Concurrent documentation can be ordered by contacting the Concurrent Software Support Center. The toll-free number for calls within the continental United States is 1-800-245-6453. For calls outside the continental United States, the number is 1-954-283-1822 or 1-305-931-2408.

Additionally, the manuals listed above are available:

- online using the PowerWorks Linux Development Environment utility, **`nhelp`**
- in PDF format in the **`documentation`** directory of the PLDE Installation CD
- on the Concurrent Computer Corporation web site at www.ccur.com

3.0. Prerequisites

Prerequisites for Concurrent C/C++ Version 5.2 for both the host system and target system are as follows:

3.1. Host System

3.1.1. Software

- Red Hat® Linux*
- Required capabilities

NOTE

The following capabilities are normally installed as part of the standard installation of Red Hat Linux and the PowerWorks Linux Development Environment. During installation of the PLDE, the user will be notified if required capabilities do not exist on the Linux system.

- PowerWorks Linux Development Environment

Capabilities	RPMs providing these capabilities
plde-HyperHelp **plde-HyperHelp-scripts** **plde-nhelp** **plde-pmax-crossdev**	**plde-x11progs-6.4.2-000** **plde-HyperHelp-scripts-6.4.2-000** *any or all of the following:* **plde-pmax-crossdev-4.3-P5-1** **plde-pmax-crossdev-5.0-SR1-1**

User applications built with Concurrent C/C++ may require other capabilities which are provided by additional RPMs included on the PLDE Installation CD. Refer to the section titled "Cross-Development Libraries and Headers" in the *PowerWorks Linux Development Environment Release Notes* (0898000) for more information.

* This product has been extensively tested on Red Hat Linux 6.1 and 6.2. However, this product has not been tested with versions of Linux supplied by other vendors.

- Red Hat Linux

Capabilities	RPMs providing these capabilities
/bin/sh **ld-linux.so.2** **libc.so.6** **libc.so.6(GLIBC_2.0)** **libc.so.6(GLIBC_2.1)** **libnsl.so.1** **rpm >= 3.0.3**	*Red Hat 6.1:* **bash-1.14.7-16** **glibc-2.1.2-11** **rpm-3.0.3-2** *Red Hat 6.2:* **bash-1.14.7-22** **glibc-2.1.3-15** **rpm-3.0.4-0.48**

3.1.2. Hardware

- an Intel®-based PC - 300Mhz or higher (recommended minimum configuration)
- 64MB physical memory (recommended minimum configuration)

3.2. Target System

3.2.1. Software

- PowerMAX OS 4.3 or later

3.2.2. Hardware

- Computer Systems:

  Power Hawk™ 620 and 640

  Power Hawk 710, 720 and 740

  PowerStack™ II and III

  Night Hawk® Series 6000

  TurboHawk™

  PowerMAXION™

- Board-Level Products:

  Motorola® MVME2604

  Motorola MVME4604

4.0. System Installation

Installation of the host portion of Concurrent C/C++ is normally done as part of the general installation of the PowerWorks Linux Development Environment software suite. A single command installs (or uninstalls) all software components of the PLDE, as described in the *PowerWorks Linux Development Environment Release Notes* (0898000).

The following section describes how to install (or uninstall) Concurrent C/C++ separately from the PLDE suite for those rare cases when this is necessary.

In addition, it is necessary to install the PowerMAX OS cross-development libraries on your Linux system in order to cross-compile and cross-link on that system.

4.1. Separate Host Installation

In rare cases, it may be necessary to install (or uninstall) Concurrent C/C++ independent of the installation of the PowerWorks Linux Development Environment software suite. This may be done using the standard Linux product installation mechanism, **`rpm`** (see **`rpm(8)`**).

The names of the RPMs associated with Concurrent C/C++ 5.2 are:

```
plde-c++-5.2
plde-c++invoker
plde-c++help-5.2
```

and the files associated with these RPMs, respectively, are:

```
plde-c++-5.2-003-1.i386.rpm
plde-c++invoker-5.2-003.i386.rpm
plde-c++help-5.2-003-1.i386.rpm
```

which can be found in the **`linux-i386`** directory on the PowerWorks Linux Development Environment Installation CD.

NOTE

The package **`plde-c++help-5.2`** contains the online manuals for the C/C++ compiler. The installation of **`plde-c++help-5.2`** is not necessary for the proper operation of the compiler.

NOTE

The user must be root in order to use the **`rpm`** product installation mechanism on the Linux system.

To install the Concurrent C/C++ RPMs, issue the following commands on your Linux system:

1. Insert the PowerWorks Linux Development Environment Installation CD in the CD-ROM drive

2. Mount the CD-ROM drive (assuming the standard mount entry for the CD-ROM device exists in **`/etc/fstab`**)

   ```
   mount /mnt/cdrom
   ```

3. Change the current working directory to the directory containing the Concurrent C/C++ RPMs

   ```
   cd /mnt/cdrom/linux-i386
   ```

4. Install the RPMs

   ```
   rpm -i plde-c++invoker-5.2-000.i386.rpm \
          plde-c++-5.2-000-1.i386.rpm \
          plde-c++help-5.2-000-1.i386.rpm
   ```

   By default, the product is installed in **`/usr/opt`**. To install in a different directory, issue the following commands instead:

   ```
   rpm -i plde-c++invoker-5.2-000.i386.rpm
   rpm -i --relocate /usr/opt=directory \
          plde-c++-5.2-000-1.i386.rpm \
          plde-c++help-5.2-000-1.i386.rpm
   ```

   where *directory* is the desired directory.

5. Change the current working directory outside the **`/mnt/cdrom`** hierarchy

   ```
   cd /
   ```

6. Unmount the CD-ROM drive (otherwise, you will be unable to remove the PowerWorks Linux Development Environment Installation CD from the CD-ROM drive)

   ```
   umount /mnt/cdrom
   ```

NOTE

The optional **`plde-c++cfront`** package containing USL iostream and complex libraries should be installed after the **`plde-c++`** package.

To uninstall the Concurrent C/C++ RPMs, use the following command:

```
rpm -e plde-c++help-5.2 \
       plde-c++invoker \
       plde-c++-5.2
```

4.2. Cross-Development Libraries

In order to cross-compile and cross-link on the Linux system, it is necessary to have certain PowerMAX OS libraries installed on that system.

The names of the RPMs containing the PowerMAX OS libraries minimally required for cross-linking are:

```
plde-pmax-crossdev-4.3
plde-pmax-crossdev-5.0
```

and are used when linking for a PowerMAX OS 4.3 or PowerMAX OS 5.0 target system, respectively. The files associated with these RPMs are:

```
plde-pmax-crossdev-4.3-P5-1.i386.rpm
plde-pmax-crossdev-5.0-SR1-1.i386.rpm
```

NOTE

The version number is part of the name of the RPM. Because of that, it is possible to install both RPMs on the Linux system at the same time. This allows the user to generate executables for multiple PowerMAX OS versions from the same Linux system.

User applications built with Concurrent C/C++ may require other capabilities which are provided by additional RPMs included on the PLDE Installation CD. Refer to the section titled “Cross-Development Libraries and Headers” in the *PowerWorks Linux Development Environment Release Notes* (0898000) for more information.

5.0. Using Concurrent C/C++ with the PLDE

The following should be taken into consideration in order to use Concurrent C/C++ with the PowerWorks Linux Development Environment.

5.1. Invoking the Compiler

On Linux systems, the **cc**, **c++**, **gcc**, and **g++** commands invoke the native Linux compilers, which are completely unrelated (and incompatible at the object level) with PowerMAX OS and the Concurrent C/C++ cross-compiler.

To utilize the Concurrent C/C++ compiler, specify the following in your PATH environment variable:

```
PATH=$PATH:/usr/ccs/bin
```

The compiler should then be invoked with either **ec** or **ec++**.

However, if you wish to be able to invoke the Concurrent C/C++ compiler as **cc** or **c++**, insert the following at the head of your PATH environment variable:

```
PATH=/usr/ccs/crossbin:$PATH
```

The **/usr/ccs/crossbin** directory contains commands named **cc** and **c++** which invoke the Concurrent C/C++ compiler as opposed to the Linux compilers.

See “Makefile Considerations” on page 10 below for more information.

5.2. Include Files and Libraries

By default, the Concurrent C/C++ compiler automatically looks for PowerMAX OS include files and libraries in the tree rooted as:

/pmax/os/*version*/*arch*

where *version* and *arch* indicate the PowerMAX OS version and target architecture of your choice (see “OS Versions and Target Architectures” on page 9 for more details).

Files located under **/usr/include** and **/usr/lib** are native Linux files and are unrelated and incompatible with the corresponding files for PowerMAX OS. Do not attempt to utilize files from those directories when building PowerMAX OS programs.

Remove any explicit references to these directories in:

- source files (e.g. `#include "/usr/include/unistd.h"`)
- Makefiles (e.g. **cc -I/usr/include**)
- build scripts

Include file references of the form:

```
#include <unistd.h>
```

or

```
#include "unistd.h"
```

need not be changed. These forms are supported, as the appropriate **/pmax/os/***version***/***arch* trees are searched.

5.3. OS Versions and Target Architectures

The PowerWorks Linux Development Environment supports building PowerMAX OS programs for various versions of PowerMAX OS and various systems.

The current versions of PowerMAX OS (**osversion**) that are supported are:

- **4.3**
- **5.0**

The current architectures (**arch**) that are supported are:

- **nh**
- **moto**
- **synergy**

which correspond to the following systems:

System type	*architecture*
PowerMAXION-4	**nh**
PowerMAXION	**nh**
Night Hawk 6800	**nh**
Night Hawk 6800 Plus	**nh**
TurboHawk	**nh**
Power Hawk 610	**moto**
Power Hawk 620	**moto**
Power Hawk 640	**moto**
PowerStack	**moto**
PowerStack II	**moto**
Power Hawk 710	**synergy**
Power Hawk 720	**synergy**
Power Hawk 740	**synergy**

NOTE

The default OS version is currently **4.3** and the default target architecture is **nh**.

You can change the **osversion** and **arch** settings in several ways:

- Specify the options on the **ec** or **ec++** command line:

  ec -o main main.c --arch=*arch* **--osversion=***os*

- Change the default for your user on a specific Linux system using the Concurrent C/C++ command line utility **c.release**:

  c.release -arch *arch* **-osversion** *os*

- When using the Concurrent C/C++ PDE utilities (**c.build**, etc.), you can:
  - Set the **arch** and **osversion** for an environment using **c.mkenv**:

    c.mkenv -arch *arch* **-osversion** *os*

  - Set the **arch** and **osversion** for a specific partition using **c.partition**:

    c.partition -oset "--arch=*arch* **--osversion=***os***" main**

5.4. Shared vs. Static Linking

By default, the Concurrent C/C++ compiler links with shared libraries.

Thus, if you attempt to execute your C++ program on a PowerMAX OS system it will require, at a minimum, the shared library **libCruntime.so**.

If your PowerMAX OS system doesn't have either the Concurrent C/C++ product or the **c++runtime** package installed, your program will fail to execute.

You can install the full PowerMAX OS version of the Concurrent C/C++ compiler, install just the **c++runtime** package, or relink your program using static libraries.

The PowerMAX OS **c++runtime** package is included on the PowerWorks Linux Development Environment Installation CD. See the section titled “Target Installation” in the *PowerWorks Linux Development Environment Release Notes* (0898000) for installation instructions.

To link your program using static libraries, append the **-Zlink=static** option to your command line:

```
ec++ -o main main.c -Zlink=static
```

5.5. Makefile Considerations

Makefiles may already contain references to **cc** or **c++** commands explicitly within them. Additionally, if default rules for compilation, such as

```
.c.o:
```

or

```
.cc.o:
```

are not explicitly mentioned, the **make** processor will also attempt to invoke **cc**, **c++**, or even **g++**.

By default, unless you have **/usr/ccs/crossbin** early in your PATH variable, these situations will result in the Linux native compilers being invoked instead of the Concurrent C/C++ compiler.

To resolve these problems you can take any of the following approaches.

5.5.1. Explicit Modification Using ec/ec++

Ensure that **/usr/ccs/bin** is in your PATH environment variable.

Modify all occurrences of **cc** and **c++** to utilize **ec** and **ec++**, respectively.

Supply default `.c.o` rules (and the like) to explicitly utilize the **ec** and **ec++** commands.

5.5.2. Use of /usr/ccs/crossbin in PATH Environment Variable

Put **/usr/ccs/crossbin** at the head of your PATH environment variable.

This will cause references to **cc** and **c++** to invoke the Concurrent C/C++ compiler as opposed to the Linux compilers.

5.5.3. Use of CC Environment or Make Variables

If you don't want **/usr/ccs/crossbin** early on your PATH (perhaps because you plan to build for Linux and/or PowerMAX OS at various times), then you'll want to just use the **ec** and **ec++** when you want to compile for PowerMAX OS (it is still necessary to add **/usr/ccs/bin** to your PATH).

One approach to using **ec** and **ec++** that requires minimal changes to Makefiles, etc., is to use environment variables or **make** variables to control which C/C++ compiler you're using. The following commands will all build using the PLDE cross-compilers:

Short-lived environment variables:

 # CC=ec CXX=ec++ make *arguments*

make variables:

 # make *arguments* **CC=ec CXX=ec++**

Long-lived environment variables:

```
# export CC=ec
# export CXX=ec++
# make arguments
```

You can also use the long-lived environment variable approach if you intend to always build for PowerMAX OS, by adding the following to your login script (e.g. **.profile** or **.login** depending on your shell):

```
export CC=ec
export CXX=ec++
```

Or, if you prefer finer-grained control, you can add lines like the following to the top of any Makefiles that should use the Concurrent C/C++ cross-compiler:

```
CC=ec
CXX=ec++
```

The changes will then only affect the modified Makefiles. Note that this solution only works for Makefiles that use the default `.c.o` and `.cpp.o`, etc. rules. If they contain hard-coded references to **cc** or **cc++**, then either **/usr/ccs/crossbin** must be used, or the Makefiles must be changed to use `$(CC)` and `$(CXX)` instead. If the Makefile references anything like **g++** (Linux's GNU C++ compiler), then it will need to be changed, regardless.

Here are two more complete and robust sets of variables which will work equally well with well-written Makefiles.

```
CC=/usr/ccs/crossbin/cc
CXX=/usr/ccs/crossbin/c++
AS=/usr/ccs/crossbin/as
AR=/usr/ccs/crossbin/ar
LD=/usr/ccs/crossbin/ld
```

Or, alternatively:

```
CC=/usr/ccs/bin/ec
CXX=/usr/ccs/bin/ec++
AS=/usr/ccs/bin/as.pmax
AR=/usr/ccs/bin/ar.pmax
LD=/usr/ccs/bin/ld.pmax
```

These two sets are mentioned in order to provide very easy support for those users that want to compile only for PowerMAX OS (**`/usr/ccs/crossbin`**) and for those users that may want to compile for either Linux or PowerMAX OS, depending on the application (**`/usr/ccs/bin`**).

6.0. Changes in This Release

6.1. Documentation Updates

The *Concurrent C/C++ Reference Manual* has been greatly expanded and now is available online through the **nhelp** and **c.man** tools. The traditional man pages have been abbreviated to only provide brief descriptions of the options, and the user is directed to the Reference Manual for complete documentation.

6.2. Program Development Environment

6.2.1. New in Release 5.1

Release 5.1 began providing a set of tools (beginning with the prefix "**c.**") for managing the building of complex programs. These tools are analogous to the "**a.**" tools in the MAXAda™ Program Development Environment. Although there isn't an exact one-to-one correspondence in the concepts, users already familiar with the "**a.**" tools will find the transition to using "**c.**" tools for C/C++ programming nearly effortless.

Release 2 and later of NightBench provides a graphical interface to the program development environments for both C/C++ and Ada.

6.2.2. New and Changed in Release 5.2

The following new commands and options have been added. Refer to *Concurrent C/C++ Reference Manual* for details.

c.build -o — New option for temporarily specifying a different output file for linking or archiving a partition.

c.build — Sets the environment variable PDE_BUILD_OPTIONS so that **make(1)** commands invoked by **c.build** can recursively invoke **c.build** with appropriate options such as verbosity options and internal options that interact with NightBench.

Creates backups of the internal PDE database. Maintains three such backups. Use the **c.restore** command to restore the environment to a backed-up state.

c.chmod — Extended to support any access mode syntax supported by **chmod(1)** and options added to further control what files are modified.

c.demangle — New tool to transform "mangled" symbol names to non-"mangled" C++ names.

c.grep — New tool to search the source behind specified compilation units, archives, executables, etc.

c.install New options to set system wide defaults for cross compiler targets:

-osversion *osversion*
-arch *architecture*
-target *microprocessor*

See the **c.release** command to see what *osversions* and *architecture*s are available for cross-targeting. See the **--target** compile option documentation for the list of available microprocessor targets. These options should only be used with the PowerWorks Linux Development Environment currently. A future PowerMAX OS release may also support cross compiling to other versions and architectures. On PowerMAX OS, the compiler will target the host systems *osversion*, *architecture* and *microprocessor*.

c.intro -o New option for specifying the path to the object file of a compilation unit.

c.make New tool to generate a **Makefile** from a program development environment.

c.options Bug fixed for compile options which take filename arguments. Paths are normalized to be relative to the environment.

The **--auto_instantiation** compile option is no longer implicitly set. The user must explicitly request automatic template instantiation, usually by setting this option in the permanent default option list.

c.options -make
New option to specify **Makefile** to be run before compiling the unit's source file.

c.options -source
New option to make it easier to specify options associated with source files, such as **-make**.

c.partition Object file partitions may specify multiple object files now. These are linked together using the **-r** option in **ld(1)**. Many link options may now be used on object partitions.

Additional link options (**-c**, **-f**, **-v**, and **--14**) are supported on archive partitions.

c.release Now sorts the release names when they are listed. If cross compilation is available (see the **c.install** command above), **c.release** will also list available *osversions* and *architecture*s, and indicate the defaults. The **-osversion**, **-arch**, and **-target** options are provided to set user specific defaults. These options should only be used where cross compiling is available, currently only under the PowerWorks Linux Development Environment.

c.restore	New tool to recover a backed-up database in a program development environment.
c.script	Generated script is much more efficient. Several options added to control format of generated script. The generated script now accepts several options: **-env**, **-f**, **-H**, and **-rel**.

6.2.3. Automatic Template Instantiation

Template instantiation is a complex issue. We have been improving support for it, but it remains intrinsically confusing. This section will attempt to explain some of the issues surrounding template instantiation and its automation. It also documents enhancements that have been made in the 5.1 and 5.2 releases to better automate template instantiation.

The compiler can't know which templates will need to be instantiated in a program until link time. Therefore, when first compiling source files, no templates are instantiated unless their instantiation is explicitly requested in the source (or via command line options). The template instantiations that can be provided and that are needed by each compilation are recorded in a **.ti** file that is placed in the same directory as the generated object file if the file is compiled with the **--auto_instantiation** option. Before actually linking, a tool called **prelink** collects a list of all the template instantiations that are needed to link successfully, and assigns each to a compilation that can provide the instantiation if the link is performed with the **--auto_instantiation** option. These assignments are recorded in **.ii** files in the same directory as the **.ti** and object files. It then recompiles those compilations and finally the linker is invoked.

6.2.4. The Problem

There is one obvious, huge, problem with this scheme. That is, if the build procedure moves the object file elsewhere, such as into an archive, then the prelinker has no way of finding the **.ti** and **.ii** files. The 5.1 release improved this situation **if** you use the program development environment tools (PDE tools). This is because the PDE's database knows how the archives were built, so it can assign a template instantiation to an compilation unit in an archive and update the archive before linking the program that uses the archive.

Consider the following **Makefile**:

```
pgm: main.o fg.a
        ec++ --auto_instantiation -o pgm main.o fg.a

main.o: main.c
        ec++ -c --auto_instantiation main.c

fg.a: f.o g.o
        ar r fg.a f.o g.o

f.o: f.c
        ec++ -c --auto_instantiation f.c

g.o: g.c
        ec++ -c --auto_instantiation g.c
```

If **f.o** and **g.o** require templates to be instantiated and **main.o** cannot provide the instantiations, then this program cannot be built because **prelink** does not know where to find the **f.ti**, **f.ii**, **g.ti**, and **g.ii** files.

6.2.5. Solving the Problem with **--prelink_objects**

The **--prelink_objects** option of **ec++** can be used to resolve the template instantiations needed by a subset of the object files. To use this in the above example, one would change the **fg.a** target to read:

```
fg.a: f.o g.o
        ec++ --prelink_objects f.o g.o
        ar r fg.a f.o g.o
```

This will direct the compiler to determine what templates are needed by **f.o** and **g.o**, assign instantiations of them to one or the other of them, if possible, and recompile them with those instantiations. There is a drawback however. Consider the following more complex **Makefile**:

```
pgm: main.o fg.a hi.a
        ec++ --auto_instantiation -o pgm main.o fg.a hi.a

main.o: main.c
        ec++ -c --auto_instantiation main.c

fg.a: f.o g.o
        ec++ --prelink_objects f.o g.o
        ar r fg.a f.o g.o

f.o: f.c
        ec++ -c --auto_instantiation f.c

g.o: g.c
        ec++ -c --auto_instantiation g.c

hi.a: h.o i.o
        ec++ --prelink_objects h.o i.o
        ar r hi.a h.o i.o

h.o: h.c
        ec++ -c --auto_instantiation h.c

i.o: i.c
        ec++ -c --auto_instantiation i.c
```

What happens if **f.o** and **h.o** both require the same template instantiation? When both archives are prelinked, they will both instantiate that same template, and a duplicate symbol error will occur at link time.

The solution for this is to arrange for the template instantiations to be placed in separate object files using the **--one_instantiation_per_object** option on both the compilation, prelinking and linking commands. This way, only one of the instantiations get loaded. The following **Makefile** demonstrates how this could be set up. This example also uses the **--instantiation_dir** option to specify where the instantiations are to be placed.

```
CFLAGS=--auto_instantiation --one_instantiation_per_object

pgm: main.o fg.a hi.a
        ec++ $(CFLAGS) -o pgm main.o fg.a hi.a

main.o: main.c
        ec++ -c $(CFLAGS) main.c

fg.a: f.o g.o
        ec++ --prelink_objects $(CFLAGS) f.o g.o
        ar r fg.a f.o g.o fg/*.o

f.o: f.c
        ec++ -c $(CFLAGS) --instantiation_dir=fg f.c

g.o: g.c
        ec++ -c $(CFLAGS) --instantiation_dir=fg g.c

hi.a: h.o i.o
        ec++ --prelink_objects $(CFLAGS) h.o i.o
        ar r hi.a h.o i.o hi/*.o

h.o: h.c
        ec++ -c $(CFLAGS) --instantiation_dir=hi h.c

i.o: i.c
        ec++ -c $(CFLAGS) --instantiation_dir=hi i.c
```

This will place the instantiation assigned to **f.o** in the directory **fg**, which then gets archived into the archive **fg.a**. Similarly, the instantiation assigned to **h.o** is placed in the directory **hi** and is archived into the archive **hi.a**. When we link, the linker will pick up the instantiation from one of the archives, and not from the other, so we link without multiply defined symbols.

There is, of course, a caveat even with this scheme. Consider the situation where **main.o** does not reference **f.o**, **g.o** also requires the template to be instantiated, and the template uses a file scoped static variable (bad programming practice to be sure, but perfectly legal). The template instantiation has been assigned to **f.o**, so it is going to reference **f.o**'s file scoped static variable. To do this, a "mangled" external name is created for it. Now when we link, should the instance of the template assigned to **f.o** be linked with **g.o**, then to resolve the reference to the file scoped static variable, **f.o** will also be linked in, even though **f.o** is not otherwise needed. This will generally be harmless, except that it inflates program size.

6.2.6. Solving the Problem with the PDE Tools

Using the PDE tools, one would set up this environment like this (the equivalent actions can be done through NightBench graphical user interface):

```
c.mkenv
c.options -set -default -- --auto_instantiation
c.intro main.c f.c g.c h.c i.c
c.partition -create archive -add "f g" fg.a
c.partition -create archive -add “h i” hi.a
c.partition -create exe -add main -parts “fg.a hi.a” pgm
```

The environment's database knows all about the **fg.a** and **hi.a** archives and how to build them. When **pgm** is built using the **c.build** command, **f.c**, **g.c**, **h.c**, **i.c**, and **main.c** will all be compiled, the archives will be created, then during the prelinking stage, template instantiations will be assigned to **f.c** or one of the other compilation units, they will be recompiled, the archive updated, and finally **pgm** will link without incident.

There is a caveat with not using the **--one_instantiation_per_object** option (this applies to using a **Makefile** too). If you have three executables that each use a pair of three compilation units and all three compilation units require the same template to be instantiated, there is no way to link the three programs without using **--one_instantiation_per_object**. If to link one program, the instantiation is assigned to one compilation unit, then the program that links in the other two compilation unit must force the instantiation to be assigned to one of them. This will cause either the first or the third program to get a multiply defined symbol on the template. This problem can be avoided by issuing the following command:

```
c.options -add -default -- --one_instantiation_per_object
```

6.2.7. Solving the Problem with Makefiles and the PDE Tools

Now, it is realized that for portability reasons, customers may not be willing to abandon their **Makefile**s. The 5.2 release has two enhancements to deal with this. The first is the **c.make** tool. This tool generates a **Makefile** from a PDE environment, making it possible to take a program that builds under the PDE on a Concurrent machine, and compile it elsewhere.

The second enhancement is in the invokers for **ec**, **ec++**, and **ar** that allow an existing **Makefile** (or any other program building mechanism) to build a program in the context of a PDE environment. These new invokers are activated by setting the PDE_ENVIRONMENT environment variable, or, instead, by placing a file called **.pde_environment** containing a single line of text specifying the path to the PDE environment to use in the directory where the compiler will be run.

The following is a slightly modified version of the above **Makefile**:

```
CFLAGS=--auto_instantiation --one_instantiation_per_object

pgm: main.o fg.a hi.a
        ec++ $(CFLAGS) -o pgm main.o fg.a hi.a

main.o: main.c
        ec++ -c $(CFLAGS) main.c

fg.a: f.o g.o
        ar r fg.a f.o g.o

f.o: f.c
        ec++ -c $(CFLAGS) f.c

g.o: g.c
        ec++ -c $(CFLAGS) g.c

hi.a: h.o i.o
        ar r hi.a h.o i.o

h.o: h.c
        ec++ -c $(CFLAGS) h.c

i.o: i.c
        ec++ -c $(CFLAGS) i.c
```

The explicit handling of the template instantiation object files has been removed because the PDE's database will handle all that for us automatically. The user may do the following:

```
mkdir pgm_env
mkenv -env pgm_env
export PDE_ENVIRONMENT='pwd'/pgm_env
make
```

Now, when the **Makefile** invokes **ec++** on **f.c**, the **ec++** invoker, rather than directly invoking the 5.2 version of the compiler, will instead invoke the following commands:

```
c.intro -env pgm_env -language C++ -o f.o f.c
c.options -env pgm_env  -set -- --auto_instantiation \
  --one_instantiation_per_object f
c.compile -env pgm_env f
```

Subsequent invocations of **make** will result in **c.options** being invoked only if the options have changed. Normally, only **c.compile** needs to be invoked. Similar actions occur for the compilation of **g.c** and **main.c**. When the **ar** invoker gets invoked on **fg.a** for the first time, it will do the following:

```
c.partition -env pgm_env -create archive -add "f g" \
              -o fg.a fg.a
ar r fg.a f.o g.o
```

Finally, when **pgm** is linked for the first time, the **ec++** invoker will do this:

```
c.partition -env pgm_env -create executable \
              -add "main" -parts "fg.a hi.a" -o pgm pgm
c.build -env pgm_env pgm
```

Since **c.build** knows all about how the archives were constructed, it can do the template instantiations, update the archives if needed, and link **pgm** without a problem. Note that we have not explicitly placed the template instantiation objects into the archives. The environment is handling management of them. If it was desired for the `fg.a` to be prelinked and have the object files placed in it, the target would be constructed like this:

```
fg.a: f.o g.o
        ar r fg.a f.o g.o
        c.partition -oset "--prelink_objects" fg.a
```

6.2.8. Miscellaneous Notes

Once the program has been made with the **Makefile** once, the user may either continue using the **Makefile**, switch to using **c.build**, or switch to using NightBench (the graphical interface to the PDE). Note however, that if the **Makefile** does other actions, such as generate source files, or create object using other compilers or the assembler, or invoke **ld** directly, these actions are not known by the environment, and thus won't be performed if **c.build** or NightBench is used, unless the environment is manually modified to use the **-make** option of **c.intro** and **c.options** to escape to a **Makefile** to perform arbitrary actions before building a source file.

Another advantage of using this scheme is that the user can set temporary options without modifying the **Makefile**. For example, if the user wants to turn on the debug option for just **f.c** temporarily, then he can do the following:

```
c.options -temp -set -- -g f
rm f.o
make
```

When the **c.compile** tool gets invoked, it will use this temporary setting. If **c.build** is used instead of **make** to build the program, then removing **f.o** isn't necessary since the PDE tools know that **f.o** is out of date when an option gets changed (the **make** command doesn't know this). Also the PDE tools know about all header file dependencies. When done with all the temporary settings, things can be reset to normal by issuing the command:

```
c.options -temp -clear all
```

Another advantage of the PDE environment is that NightBench can be used to examine what specific template instantiations were done and who did them. It also provides a way to manually override individual automatic decisions.

6.3. Previous Versions

If an earlier C or C++ compiler is still installed on the system, the system administrator may choose to configure the previous compiler commands (**cc**, **hc**, **cc++**, **c++**, **analyze**, and **report**) to invoke the pre-5.1 release. However, by default, they will invoke the 5.2 release. The user may override this default behavior by setting the **PDE_RELEASE** environment variable to either **pre5.1**, **5.2**, or another release name, or by using the **c.release** command to select the user-specific default release.

6.4. --preinclude

The **--preinclude** command-line option can be used to cause inclusion of a specified source file before the primary source file is read.

6.5. -arch Link Option

The **-arch** link option partially determines which libraries are included and/or referenced when linking partitions. Its syntax is:

-arch *architecture*

where *architecture* is the target architecture, and can be one of: **nh**, **moto**, or **synergy**.

If specified for a particular partition or for its containing environment, the partition is linked so that it will execute properly on the specified architecture.

NOTE

If building on a Linux system with **plde-pmax-crossdev** versions capable of supporting multiple architectures, the **-arch** option is *required* to be able to link.

To avoid having to specify the architecture for every partition, it is possible to specify it for an entire environment via the environment-wide link options (see "Environment-wide Link Options" on page 22). This can be done when creating the environment via:

c.mkenv ... -arch *arch* **...**

which is a shorthand for:

c.mkenv ... -oset "-arch *arch***" ...**

The PowerMAX OS cross-development libraries (e.g. **plde-pmax-crossdev-4.3** and/or **plde-pmax-crossdev-5.0**) must be installed on the Linux system. Refer to "Cross-Development Libraries" on page 7 as well as the section titled "Cross-Development Libraries and Headers" in the *PowerWorks Linux Development Environment Release Notes* (0898000) for more information about installing the PowerMAX OS cross-development libraries on your Linux system.

6.6. -osversion Link Option

The **-osversion** link option partially determines which libraries are included and/or referenced when linking partitions. Its syntax is:

-osversion *ver*

where *ver* is the target PowerMAX OS version (e.g. 4.3 or 5.0).

NOTE

If building on a Linux system with multiple **plde-pmax-crossdev** versions, the **-osversion** option is *required* to be able to link.

To avoid having to specify the OS version for every partition, it is possible to specify it for an entire environment via the environment-wide link options (see “Environment-wide Link Options” on page 22). This can be done when creating the environment via:

```
c.mkenv ... -osversion ver ...
```

which is a shorthand for:

```
c.mkenv ... -oset "-osversion ver" ...
```

The PowerMAX OS cross-development libraries (e.g. **`plde-pmax-crossdev-4.3`** and/or **`plde-pmax-crossdev-5.0`**) must be installed on the Linux system. Refer to “Cross-Development Libraries” on page 7 as well as the section titled “Cross-Development Libraries and Headers” in the *PowerWorks Linux Development Environment Release Notes* (0898000) for more information about installing the PowerMAX OS cross-development libraries on your Linux system.

6.7. Environment-wide Link Options

Concurrent C/C++ 5.2 introduces the concept of environment-wide link options. Environment-wide link options affect all partitions within an environment.

Environment-wide link options may be specified when the environment is created using the **`-oset`** *opts* option to **`c.mkenv`**.

Link options that affect all the partitions in the entire environment may also be specified using the **`-default`** option to **`c.partition`** in combination with the **`-o`***commands* (**`-oset`**, **`-oappend`**, **`-oprepend`**, **`-oclear`**).

For example:

```
c.partition -default -oset -s
```

sets the environment-wide link options to **`-s`**.

To list the environment-wide link options, issue:

```
c.partition -default
```

by itself.

6.8. Call of virtual function during subobject construction

Calls of virtual functions during subobject construction in the presence of virtual base classes now always get to the correct functions.

6.9. Default arguments

Default arguments on template functions and on member functions of template classes are now instantiated only when needed.

6.10. Class name injection

The name of a class is injected as a member of the class. Controlled by **`--[no_]class_name_injection`**.

6.11. Argument-dependent (Koenig) lookup on function calls with the standard f(x,y,z) syntax

The name of the function is looked up in the namespaces and classes associated with the types of the arguments. Controlled by **`--[no_]arg_dep_lookup`**.

6.12. Non-injection of friend declarations

Functions declared in friend declarations in classes are no longer entered into the innermost enclosing namespace scope. They are after a fashion still declared in those scopes, but the name is not visible unless it is actually declared by some other (non-friend) declaration. Such friend functions are found by argument-dependent lookup. This feature is off by default in the release, because it has shown to break some real-world code. Controlled by **`--[no_]friend_injection`**.

6.13. String literals

String literals have `const` type. Controlled by **`--[no_]const_string_literals`**.

6.14. Return statement

A `return` statement in a `void` function can now return a `void` expression.

6.15. extern “C”

Support for `extern "C"` and its interaction with namespaces has been improved.

6.16. Universal character names

Universal character names (e.g., `\u0401`) are now accepted in C++ mode in identifiers, characters literals, and string literals.

6.17. Include file suffixes

There is now a mechanism that will automatically add a suffix on a `#include` of a name without a suffix (e.g., `#include <string>`). The list of suffixes to be tried may be specified with the `--incl_suffixes` option. The default setting causes the compiler to look first for a file without a suffix, then with a “`.h`” suffix, then with a “`.hpp`” suffix.

6.18. Class layout

Class layout is not optimized to avoid allocating space for empty base classes.

6.19. C99 features

Designated initializers and variadic macros from the C99 standard have been implemented. These features are disabled by default.

6.20. Template template parameters

Template template parameters are not implemented. For example

```
template <template <class X> class T> struct A {
   T<int> ti;
};
template <class T> struct B {};
template <template <class X> class T,
          class T2> void f(T<T2>);
template <template <class X> class T>
   void g(A<T>);
int main() {
   A<B> ab;
   g(ab);
   B<int> bi;
   f(bi);
}
```

6.21. Members of unnamed namespaces

The members of unnamed namespaces now have external linkage.

6.22. Pointers-to-members

Casts between unrelated pointers-to-members are now accepted.

6.23. Function-like macros

It is now possible to define function-like macros on the command line, e.g. "**`-Df(x)=x`**".

6.24. Additional Pragmas

The following pragmas have been added:

- `align`
- `cautions`
- `errcount`
- `error`
- `min_align`
- `opt_class`
- `opt_level`
- `optimize_for_space`
- `optimize_for_time`
- `pack`
- `warnings`

6.25. AltiVec Support

Motorola defined several extensions to C and C++ for accessing the AltiVec instructions of the PowerPC 7400 family of microprocessors. These extensions include new keywords, intrinsic functions, runtime functions, and a pragma. See the Motorola document *AltiVec Technology Programming Interface Manual* for detailed documentation of these extensions. This document will only discuss issues specific to Concurrent's implementation.

Currently the only microprocessor supported by Concurrent's C/C++ compiler that has AltiVec instruction is the MPC7400. It may be selected by the **`--target=ppc7400`** (or **`--target=mpc7400`**) option. See discussion of **`c.install`** and **`c.release`** commands for ways of setting the default target microprocessor if none is specified.

6.25.1. New Keywords for AltiVec

The following table shows the keywords that were added for AltiVec support and the compiler options to enable and disable them (note that **`bool`** is present in standard C++).

Keyword	Option to Enable	Option to Disable
`vector`	**`--vectors`**	**`--no_vectors`**
`pixel`	**`--vectors`**	**`--no_vectors`**
`bool`	**`--bool`**	**`--no_bool`**
`__vector`		N/A
`__pixel`		N/A

The following table documents the default settings for those compiler options:

C & target microprocessor has AltiVec instructions	C & target microprocessor does not have AltiVec instructions	C++
--vectors	**--no_vectors**	**--no_vectors**
--bool	**--no_bool**	**--bool**

6.25.2. New Intrinsic Functions for AltiVec

To access the intrinsic functions for AltiVec, include the header file **<altivec.h>**.

6.25.3. New Pragma for AltiVec

`#pragma altivec_vrsave` *{* `on` | `off` | `allon` | `allzero` *}*

The allzero option will set the VRSAVE register to zero in the procedure in which it is used. Refer to Motorola documentation for the other options.

6.25.4. varargs/stdarg for AltiVec

The base 5.2 release of Concurrent C/C++ does not support **varargs**/**stdarg** for vector types. A future release or patch of the header files and of the compiler (both require changes) will provide support for this feature.

6.25.5. Runtime for AltiVec

The Motorola defined `vec_malloc()`, `vec_calloc()`, `vec_realloc()`, and `vec_free()` are not implemented. The standard system allocation routines -- `malloc()`, `calloc()`, `realloc()`, and `free()` respectively -- return quadword aligned memory (except for `free()` obviously), and should be used instead.

6.25.6. Interoperability with Non-AltiVec for AltiVec

The global variable `__vectors_present` allows the user to determine at runtime whether the system on which the program is running supports the AltiVec instruction set. Attempting to execute an AltiVec instruction on a system that does not support it will result in an Illegal Instruction exception. Any procedure that contains AltiVec instructions may have additional AltiVec instructions generated in the routines prolog and epilog.

In order to provide both normal and AltiVec accelerated versions of a routine, the following code sequence is recommended:

```
extern int __vectors_present;

type vector_function(arguments) {
   // Vector implementation of function
   ...
}

type function(arguments) {
   if (__vectors_present) {
      return vector_function(arguments);
   }

   // Non-vector implementation of function
   ...
}
```

Now, by calling `function()`, if the program is running an an AltiVec-supporting system, the program will execute the AltiVec accelerated version of the function. But if the program is run on a system that doesn't support AltiVec instructions, the program will not execute any AltiVec instructions.

7.0. Cautions

7.1. PowerMAX OS and Linux Compatibility

The Concurrent C/C++ component of the PowerWorks Linux Development Environment is a cross-compilation system. The object files, archives, and shared libraries produced by it are not compatible with Linux and will not run in that environment. In general, Linux binary utilities (e.g. **`binutils`**) will be unable to interpret them.

Object files, archives, and shared libraries produced by Concurrent C/C++ are fully compatible with similar objects produced on PowerMAX OS systems.

However, the compilation environments themselves cannot be shared between PowerMAX OS and Linux systems, due to their internal representations (byte ordering).

For example, if two environments were created using PowerMAX OS and Linux versions of Concurrent C/C++, attempts to share the environments (such as adding the PowerMAX OS environment to the Environment Search Path of the Linux environment using **`c.path -i`** command) would result in an internal error.

7.2. Retain Source

Users are encouraged to retain the source for their applications. Major releases may have changes in the object-file format which will require the recompilation of their programs. This release is one such release. In the process of implementing additional ANSI/ISO C++ features, some changes to "name mangling" and interfaces to runtime routines were necessary.

7.3. <curses.h> and bool

The **`<curses.h>`** header file contains a **`bool`** type definition. In compilation modes where **`bool`** is a keyword, this results in a compilation error. There are two workarounds. The user may turn off the **`bool`** keyword with the **`--no_bool`** option, or he may use the following sequence when using the header file:

```
#define bool _curses_bool
#include <curses.h>
#undef bool
```

When using this second workaround, if the user makes reference to curses' `bool` type definition, the user must use the name **`_curses_bool`** instead.

7.4. Structure Compares

The C 4.3 and earlier releases supported an enhancement that allowed the user to implicitly compare a structure to zero (meaning all bytes of the structure are zero) in an if or while statement:

```
struct S {int a,b,c,d,e,f;} s;
...
if (s) {
...
```

As this enhancement is not present in any other major C compiler and is undocumented in the Concurrent C compiler, it is no longer present in C/C++ 5.1. The user can easily write a small function to test a structure for being zero should any code actually use this feature.

7.5. Known Problem with C/C++ Standard Library

The Concurrent software workaround provided below addresses a problem detected by the C++ standards committee Library Working Group. Concurrent will continue to monitor the situation and evaluate the group's final recommendation for any further action that might be necessary.

If a function is explicitly instantiated and its type is supplied, the compiler looks up all functions of that name and tries to instantiate in an attempt to determine what the closest match is. If one of these cannot be instantiated for the given type, the compiler will then issues an error.

The workaround is to omit the explicit type, but let the compiler determine it from context. An example is provided below:

```
namespace std {
 template <class Iterator> struct iterator_traits {
   typedef typename Iterator::difference_type difference_type;
 };

 template <class T> class reverse_iterator;

 template <class T>
   void operator+(typename iterator_traits<T>::difference_type,
            const reverse_iterator<T>&);

 template <class T> struct complex;

 template <class T> void operator+(const T& lhs, const complex<T>& rhs){;}

}

// Explicit instantiation which gives the error:
template void std::operator+<float>(const float&, const std::complex<float>&);

// Workaround which does not:
template void std::operator+<>(const float&, const std::complex<float>&);
```

7.6. Pragma min_align

Using a pragma `min_align` of the form:

```
#pragma min_align [struct | union | class] name align
[struct | union | class] name {
  declaration
};
```

will result in a front end assertion violation.

The workaround is to declare the `[struct | union | class]` *name* previous to the `#pragma` statement.

Specifically:

```
struct foo_tag;
#pragma min_align foo_tag 16
struct foo_tag {
   double b1;
   double b2;
   double b3;
} foo;
```

will work while:

```
#pragma min_align foo_tag 16
struct foo_tag {
   double b1;
   double b2;
   double b3;
} foo;
```

will give an assertion violation.

8.0. Direct Software Support

Software support is available from a central source. If you need assistance or information about your system, please contact the Concurrent Software Support Center at 1-800-245-6453. Our customers outside the continental United States can contact us directly at 1-954-283-1822 or 1-305-931-2408. The Software Support Center operates Monday through Friday from 8 a.m. to 7 p.m., Eastern Standard time.

Calling the Software Support Center gives you immediate access to a broad range of skilled personnel and guarantees you a prompt response from the person most qualified to assist you. If you have a question requiring on-site assistance or consultation, the Software Support Center staff will arrange for a field analyst to return your call and schedule a visit.