3.3 Release Notes: Precompiled Headers

This file contains release notes for the 3.1, and 3.0 releases of Precompiled Headers. Items specific to the 3.1 release are listed first, and the Release 3.0 notes follow. There are no items specific to the 3.3 or 3.2 releases.

A precompiled header is a C or Objective C header file that has been preprocessed and parsed, thereby improving compile time and reducing symbol table size. The macros and external declarations from the original header are sorted to enable fast lookup. Precompiled headers reduce the amount of information processed and output by the compiler. They can also serve to reduce link times and executable size when symbol table information is kept. A new implementation of the C preprocessor (cpp) can use precompiled headers in place of standard headers.

In most cases, the use of precompiled headers is transparent. Precompiled headers are simple enough to use that most projects require no conversion at all, or can be converted in a day or less.

Notes Specific to Release 3.1

Fat Precompiled Headers

Due to the additional capability of a precompiled header to support multiple architectures in Release 3.1, *precompiled headers built using Release 3.0 will not be usable under 3.1*. To re-generate your precompiled headers, usually a `make clean all' will suffice (for more details, see **Creating Your Own Precompiled Headers** below).

When precompiling with multiple **-arch** flags in a multiple architecture environment, precompiled headers will be generated ªfatº. This makes them usable by builds running on more than one architecture. The precompiled system headers on the 3.1 CD-ROM will be shipped ªfat.º

For more details on multiple- and cross-architecture compilation, see the Compiler and CompilerTools release notes.

Known Problems

Reference: 31148

Problem: The **index()** macro in **string.h** can cause precompiled headers not to be used.

Description: Include file dependencies occasionally cause conflicts with the current precompiled header format, causing the precompiled header not to be used. This will slow down compilation, but won't cause any errors.

Workaround: Make sure the **#include** or **#import** lines for precompiled headers precede all others wherever possible.

Notes Specific to Release 3.0

Using Precompiled System Headers

The precompiled version of a header file has a ª.pº extension, rather than the standard ª.hº extension. You should *not* refer to **appkit.p** in your source files; just use **appkit.h** and the cpp-precomp preprocessor will use the precompiled form if it's available and appropriate.

When the preprocessor encounters an include directive, it automatically looks for a precompiled version of the header. If one is found, it checks whether the context is equivalent to the context in which the precompiled header was built;if it is, the precompiled header is used. However, if any of the following problems occur, the non-precompiled form is included instead:

- A header which was included by the precompiled header could not be found in the filesystem to verify its modification time, or the modification time did not

match. In practice, this never occurs for precompiled headers that are part of the release, and occurs only rarely when programmers build their own precompiled headers.

- A macro was defined when the precompiled header was built, but is not defined in the current context. This is only a problem if the macro was actually referenced somewhere in the precompiled header.
- A macro was undefined when the precompiled header was built, but is defined in the current context. This is only a problem if there might have been an invocation of the macro in the precompiled header.

Compile-time warnings (described at the end of this file) indicate the nature of any problems that occur. These default warnings can be turned off with the **-Wno-precomp** in cases where a precompiled header and a compilation context can't be made compatible.

If you're developing a small project, you don't need to bother building your own precompiled headers; just use the standard precompiled forms of system headers like **appkit.h** and **mach.h**. It's easy to create your own precompiled headers if you wish to do so, however, as described in the next section.

Creating Your Own Precompiled Headers

You create a precompiled header by passing the new **-precomp** switch to **cc**. Depending on the context(s) in which the header is used, **-D** switches should also be passed to **cc**, as explained below.

```
% cc -precomp foo.h -o foo.p
```

We say a header is ªcontext dependentº if the definitions in the header may change depending on the context in which it is included. Most uses of conditional compilation and macro expansions cause context dependence. For instance, the following header is context dependent:

```
#ifdef DEBUG
int a;
#else
int b;
#endif
```

The context at any point is determined by the macros that are defined there. A precompiled header must be created in a context equivalent to that where it is used. By passing switches to the preprocessor, any set of macros can be predefined, creating a context in which the precompiled header is built. This is done by passing a **-D** switch for each macro in the context.

A precompiled header built from ªsystem headersº typically requires no **-D** switches, because programmers usually include system headers in a context-independent way. For example, the public appkit headers contain almost no preprocessor conditionals; clients cannot change declarations in headers by defining macros. So the command to build a precompiled header from **appkit.h** is:

```
% cc -precomp appkit.h -o appkit.p
```

But if you must use a header **bar.h** in a context where FOO is defined, you should

build the precompiled header as follows:

```
% cc -precomp -DFOO bar.h -o bar.p
```

You should also pass any preprocessor switches, such as **-I**, that you use in your project. This can be most easily accomplished by simply precompiling a project's headers with the same $(CFLAGS) used to compile modules in that project.

By making precompiled headers bigger (that is, containing more headers), a given C file may include fewer precompiled headers, and will generally compile faster. However, the bigger a precompiled header is, the more likely that name conflicts will occur.

For example, if you were to combine all the headers for a project, including system headers, into a single precompiled header, it is conceivable that there would be a name conflict. There may be a macro defined that happens to match one of your local identifiers, or there may be a public struct declared that happens to match one of your private struct names. Such conflicts manifest themselves as preprocessing errors, syntax errors, or semantic errors. The conflicts may be resolved by renaming identifiers, or removing a conflicting header from the precompiled header.

Another disadvantage to big precompiled headers is file dependencies. If all of the C files in a project depend on a single precompiled header which in turn depends on all headers in the project, then changing a header requires recompilation of the entire project. A better approach is to build a precompiled header containing all the system headers used by a project, and perhaps also a separate precompiled header for the local headers in the project. We recommend that during development, while local headers are changing, precompiled headers be used only

for system files. When local headers have stabilized, they may be combined into a precompiled header.

A precompiled header records absolute path names for all the headers that went into it. These paths are then checked when the precompiled header is used. Therefore a precompiled header should be built in the same directory in which it is to be used, and all the headers that went into the precompiled header must not be moved or modified.

Additional Information

The cpp-precomp preprocessor is required in order to use precompiled headers, and there are several incompatibilities with the Release 2.0 preprocessor and parser. For example, preprocessing errors and syntax errors are in a slightly different format.

Only rarely will you have trouble building a precompiled header. The most common problem you might encounter is that the header doesn't parse; this is often because the header does not include other headers it depends on, so that there are undefined types. Another typical problem is conflicting definitions, which can be solved by renaming identifiers or removing a header from the precompiled header.

The following list describes the compile-time warnings that may occur when using a precompiled header:

- **could not use precompiled header `***header***.p'**

  The precompiled header could not be used for one of the reasons below.

- **`*header*.h' has different date than in precomp**

  The modification time of the header on the disk does not match the modification time of the header when the precompiled header was built.

- **macro `*macro*' defined by `*header*.h' conflicts with precomp**

  A previously included precompiled header defines a macro differently than does the current precompiled header being processed.

- **macro `*macro*' defined on command line conflicts with precomp**

  Similar to the previous warning, except that the earlier definition of the macro occurred on the command line.

- **macro `*macro*' previously defined on command line for precomp not defined**

  A macro included in a precompiled header by way of the precompilation command line is not defined in the current context, but used by the precomp.

- **macro `*macro*' previously defined on command line for precomp does not match current command line definition**

  A macro included in a precompiled header by way of the precompilation command line is not defined in the current context, but used by the precomp.

- **macro `*macro*' redefined, locations of the conflict are:**
  header1.h:23
  header2.h:47 (within the precompiled header)

The macro has been defined in two different ways in two different precompiled headers

- **#ifdef `*SYM*' not defined when precompiled, but now defined at *header*.h:23**

  A symbol was defined, at the specified location, prior to the inclusion of this precompiled header, but was not when the header was precompiled. Since this symbol is used in an **#ifdef**, the precompiled header does not contain all the source code desired by the including context.

- **could not find `*header*.h'**

  The header which was included by the precompiled header could not be found on the disk to verify its modification time.

- **could not use precomp `*header*.p' (incorrect version)**

  It was discovered that the version of the referenced precompiled header is incompatible with the compiler, possibly signifying a corrupt or obsolete *header*.p.

- **explicit reference to precompiled `*header*.p' failed**

  Although the inclusion of headers with a ª.pº suffix is discouraged due to portability considerations, it is legal to explicitly reference precompiled headers. The above error is generated if the precompiled header is not appropriate in the enclosing context.