3 PREPROCESSOR

Overview

The preprocessor program (`pp.exe`) evaluates and processes preprocessor commands in your source files. With these commands, you direct the preprocessor to define macros and symbolic constants, include header files, test for errors, and control conditional assembly and compilation.

The preprocessor is run by an assembler and linker at the operating system's command line or within the VisualDSP++ environment. These tools accept command-line switches for the preprocessor on their command lines and pass them to the preprocessor. The ADSP-21xxx DSP preprocessor can also operate from the command line with its own command-line switches. A list of preprocessor switches appears in Table 3-3 on page 3-17.

In the default mode of operations, the preprocessor reads code from an assembly source file (`.ASM`); modifies it according to preprocessor commands; and outputs an altered preprocessed assembly file (`.IS`) using the name of the source file being processed (the `-o` switch is on). When the preprocessor runs without `-o`, no intermediate file is created, and all the output is sent to the console display (standard output). The preprocessed source file is a primary input file for the assembler program; it is purged when a binary object file `(.DOJ)` is created.

The compiler also includes a preprocessor that lets you use preprocessor commands within your C/C++ source. The compiler preprocessor automatically runs before the compiler. This preprocessor is separate from the assembler and has some features that may not be used within your assembly source files. For more information, see the *VisualDSP++ 2.0 C/C++ Compiler & Library Manual for ADSP-21xxx DSPs*.

This chapter provides reference information on the preprocessor's switches and commands. The reference information available on this topics is as follows:

- "Preprocessor Guide" on page 3-3

  This section describes the preprocessor's options, which are accessible from the operating system command line or from the VisualDSP++ environment.

- "Preprocessor Command-Line Interface" on page 3-9

  This section describes the preprocessor's switches, including syntax and usage examples.

- "Preprocessor Directives/Commands" on page 3-16

  This section describes the preprocessor's directives/commands, including syntax and usage examples.

Preprocessor Guide

This section contains the preprocessor information you need to know to build your programs from a command line or from the VisualDSP++ environment. Software developers using the preprocessor should be familiar with the following operations:

- "Setting Preprocessor Options" on page 3-3
- "Using Header Files" on page 3-4
- "Writing Macros" on page 3-5
- "Writing Compound Statements As Macros" on page 3-6
- "Using Predefined Macros" on page 3-7

Setting Preprocessor Options

When developing a DSP project, you may find it useful to modify the preprocessor's default options. Because the assembler, compiler, and linker automatically run the preprocessor as your program is build (unless you skip the processing entirely), these tools can receive input for the preprocessor program and direct its operation. The way you set the preprocessor options depends on the environment used to run your DSP development software:

- From the operating system command line, you select the preprocessor's command-line switches. For more information, see the "Preprocessor Command-Line Interface" on page 3-9.
- From the VisualDSP++ environment, you select the preprocessor's options in the **Assemble**, **Compile**, and **Link** tabs of the **Project Options** dialog box, selected via the **Project** menu. For more information on these option settings, see the *VisualDSP++ 2.0 User's Guide for ADSP-21xxx DSPs* and online Help.

Using Header Files

A header file (`.H`) contains declarations and macro definitions. The `#include` preprocessor command includes a copy of the header file at the location of the command. There are two main categories of header files:

- System header files declare the interfaces to the parts of the operating system. Include them in your program for definitions and declarations that access system calls and libraries. Use angle brackets to indicate a system header file.

  Examples:

  ```
  #include <device.h>
  #include <major.h>
  ```

  System header files are installed in the `…21k/include/sys` and `…211xx/include/sys` folders.

- User header files contain declarations for interfaces between the source files of your program. Use double quotes to indicate a user header file.

  Examples:

  ```
  #include "def21060.h”
  #include "fft_ovly.h"
  ```

  User header files are installed in the `…21k/include` and `…211xx/include` folders. This directory also includes run-time library files.

For syntax information and usage examples on the `#include` preprocessor command, see “#include” on page 3-28.

Writing Macros

The preprocessor processes macros in your C, C++, and assembly source files. Macros are useful for repeating instruction sequences in your source code.

The term *macro* defines a macro-identifying symbol and corresponding definition that the preprocessor uses to substitute the macro reference(s). Macros allow text replacement, file inclusion, conditional assembly, conditional compilation, and macro definition.

Macro definitions start with `#define` and end with a carriage return unless you define them within the `do {…} while` pair, which allow macros to end with a semicolon (`;`). If a macro definition is longer than one line, place the backslash character (`\`) at the end of each line except the last. This character indicates that the macro definition continues on the next line and allows to break a long line for cosmetic purposes without changing its meaning. For more syntax information and usage examples for the `#define` preprocessor command, see "#define" on page 3-19.

Example:

```
#define false 0

#define min(a,b) ((a) < (b) ? (a):(b))

#define ccall(x) \
   r2=i6; i6=i7; \
   jump (pc, x) (db); \
   dm(i7,m7)=r2; \
   dm(i7, m7)=PC
```

Statements within the macro can be instructions, commands, or other macros. Macro nesting (macros called within another macro) is limited only by the memory that is available during preprocessing.

A macro can have arguments. When you pass parameters to a macro, the macro serves as a general-purpose routine that is usable in many different programs. The block of instructions that the preprocessor substitutes can vary with each new set of arguments. A macro, however, differs from a subroutine call. During assembly, each instance of a macro inserts a copy of the same block of instructions, so multiple copies of that code appear in different locations in the object code. By comparison, a subroutine appears only once in the object code, and the block of instructions at that location are executed for every call.

The preprocessor also supports a C9X extension to preprocessor syntax that allows creating macro definitions with a variable number of arguments. To define such a definition, you end the formal list with an ellipsis; for example, `#define foo(x,y, …)`. The error occurs when the ellipsis is not the last item in the formal list of arguments.

To invoke a macro, place a semicolon (`;`) after the macro identifying symbol, as shown in the following example:

```
#define mac mrf=mrf+r2*r5(ssfr) // macro definition
r2=r1-r0;                       // set parameters
r5=dm(i1,m0);
mac;                            // macro invocation
```

Writing Compound Statements As Macros

When writing a macro, it is sometimes useful to define it so you can treat it as a function call. By creating macros that expand this way, you make your source code easier to read and maintain. These types of macros contain multiple instructions, which are referred to as compound statements.

To write a compound statement with multiple instructions in the macro definition, terminate each instruction except the last one with a semicolon. Place a carriage return at the end of the last instruction to indicate the end of the macro.

To define C or C++ macros that do not terminate with a semicolon, enclose the macro definition within a `do … while` loop.

The following code segment demonstrates this practice with the macro `SKIP_SPACES`:

```
/* A macro written as a compound statement */
#define SKIP_SPACES (p, limit) \
do {register char *lim = (limit); \
     while (p != lim) { \
     if (*p++ != ' ') { \
          p-; break; }}}\
          while (0)
```

Enclosing the definition within the `do {…} while (0)` pair transforms the preprocessor output from a compound statement to a single statement. This lets you treat the expanded macro as a function:

```
if (*p != 0)
else …
  SKIP_SPACES(p, lim); // uses compound macro
```

For more syntax information and usage examples for the `#define` preprocessor command, see “#define” on page 3-19.

Using Predefined Macros

In addition to macros you define, some DSP development tools include predefined macros that you can use in your code.

The assembler preprocessor provide a set of predefined macros that you can use in your assembly code. The preprocessor automatically replaces each occurrence of the macro reference found throughout the program with the specified value. However, predefined macros are not expendable within comments.

The predefined macros that `pp` provides are listed and described in Table 3-1.

Table 3-1. Predefined Macros

Macro	Definition
`ADI`	The preprocessor defines `ADI` as 1.
`__LINE__`	The preprocessor defines `__LINE__` as 4.
`__FILE__`	The preprocessor defines `__FILE__` as the name and extension of the file in which the macro is defined, for example, ‘`macro.asm`’.
`__STDC__`	The preprocessor defines `__STDC__` as 1.
`__TIME__`	The preprocessor defines `__TIME__` as current time in the 24-hour format ‘`hh:mm:ss`’, for example, ‘`06:54:35`’.
`__DATE__`	The preprocessor defines `__DATE__` as current date in the format ‘`Mm dd yyyy`’, for example, ‘`Oct 02 2000`’.

Note that the `__DATE__`, `__FILE__`, and `__TIME__` macros return strings within the single quotation marks (‘ ’).

Preprocessor Command-Line Interface

Preprocessing is the first step in the process of building (assembling, compiling, linking) your programs. The preprocessor reads code from a source file (`.ASM`), modifies it according to preprocessor commands, and generates an altered preprocessed assembly (`.IS`) or C/C++ (`.I`) source file. The preprocessed source file is a primary input file for the assembler or compiler program; it is purged when the a binary object file`(.DOJ)` is created.

Running the Preprocessor

To run the preprocessor from the command line, type the name of the program followed by arguments in any order:

```
pp [-switch1[-switch2 …]] [sourceFile]
```

where:

- `pp` — Name of the ADSP-21xxx SHARC DSP preprocessor program.
- `-switch1,-switch2` — Switches to process.

  The preprocessor offers many optional switches to select its operations and modes. Some preprocessor switches take a file name as a required parameter.
- `sourceFile` — Name of the source file to process. The preprocessor supports relative and absolute path names. The `pp.exe` preprocessor outputs a list of command-line switches when runs without this argument.

If the `sourceFile` was incorrectly specified, usually because of a typing error, or a user does not have access to the file, the preprocessor fails to open the named file. That is a fatal error and the preprocessor will exit without doing any processing.

When the preprocessor runs, it modifies your source code by:

- Including system and user-defined header files
- Defining macros and symbolic constants
- Providing conditional assembly and compilation

You specify preprocessing options with preprocessor commands, lines starting with `#`. Without any commands, the preprocessor performs the following three global substitutions:

- Replaces comments with single spaces
- Deletes line continuation characters (`\`)
- Replaces predefined macro references with corresponding expansions

The following cases are notable exceptions to the described substitutions:

- The preprocessor does not recognize comments or macros within the file name delimiters of an `#include` command.
- The preprocessor does not recognize comments or predefined macros within a character or string constant.

The following command line, for example:

```
pp -Dfilter_taps=100 -v -o bin\p1.doj p1.asm
```

runs the preprocessor with:

`-Dfilter_taps=100` — Defines the macro `filter_taps` as equal to 100.

`-v` — Displays verbose information for each phase of the preprocessing.

`-o bin\p1.is` — Specifies the name and directory for the intermediate preprocessed file.

`p1.asm` — Specifies the assembly source file to preprocess.

Preprocessor Command-Line Switch Summary

Table 3-2 lists the `pp.exe` option set. Switch descriptions starts on page 3-12.

Table 3-2. Preprocessor Command-Line Switch Summary

Switch Name	Description
`-cs!`	Elide "`!`" style comments.
`-cs/*`	Elide "`/*   */`" style comments.
`-cs//`	Elide "`//`" style comments.
`-cs{`	Elide "`{  }`" style comments.
`-csall`	Elide comments in all formats.
`-Dmacro[=definition]`	Define *`macro`*.
`-h[elp]`	Output a list of assembler switches.
`-i|Idirectory`	Search *`directory`* for included files.
`-M`	Make dependencies only, do not assemble.
`-MM`	Make dependencies and assemble.
`-Mo filename`	Specify *`filename`* for the make dependencies output file.
`-Mt filename`	Make dependencies for the specified source file.
`-o filename`	Output named object file.
`-v[erbose]`	Display information about each assembly phase.
`-version`	Display version info about the assembler and preprocessor programs.

Preprocessor Command-Line Switch Descriptions

A description of each switch appears in the following sections.

-cs!

`-cs!` switch directs the preprocessor to pass the "`!`" single-line comment format.

-cs/*

`-cs/*` switch directs the preprocessor to pass the "`/* */`" multi-line comment format.

-cs//

`-cs//` switch directs the preprocessor to pass the "`//`" single-linecomment format.

-cs{

`-cs {`switch directs the preprocessor to pass the "`{}`" single-line comment format.

-csall

`-csall` switch directs the preprocessor to pass comments in all formats.

-D*macro*[=*def*]

The `-D` (define macro) switch directs the preprocessor to define a macro. If you do not include the optional definition string (`=`*`def`*), the preprocessor defines the macro as value 1.

Some examples of this switch are as follows:

```
-Dinput            // defines input as 1
```

```
-Dsamples=10        // defines samples as 10
-Dpoint="Start"     // defines point as the string “Start”
```

-h[elp]

The `-h` or -`help` switch directs the preprocessor to output to standard output the list of command-line switches with a syntax summary.

-i|I*directory*

The `-idirectory` or `-Idirectory` (include directory) switch directs the preprocessor to append the specified directory (or a list of directories separated by semicolon) to the search path for included files. These files are:

- header files (`.h`) included with the `#include` command
- data initialization files (`.dat`) specified with the `.VAR` directive

Note that no space is allowed between `-i|I` and the pathname.

The preprocessor searches for included files in the following order:

- Current directory
- `include` subdirectory of the VisualDSP++ installation directory
- Specified directory (a list of directories). The order of the list defines the order of multiple searches.

-M

The `-M` (generate make rule only) switch directs the preprocessor to output a rule, which is suitable for the make utility, describing the dependencies of the source file. The output, a make dependencies list, is written to `stdout` in the standard command-line format:

```
source_file: dependency_file.ext
```

where *`dependency_file.ext`* may be an assembly source file, a header file included with the `#include` preprocessor command, or a data file.

When the `-o` *`filename`* option is used with `-M`, the preprocessor outputs a make dependencies list to the named file.

-MM

The `-MM` (generate make rule and assemble) switch directs the preprocessor to output a rule, which is suitable for the make utility, describing the dependencies of the source file. After preprocessing, the assembly of the source into an object file proceeds normally. The output, a make dependencies list, is written to `stdout` in the standard command-line format:

```
source_file.doj: dependency_file.ext
```

where *`dependency_file.ext`* may be an assembly source file, a header file included with the `#include` preprocessor command, or a data file.

For example, the source `vectAdd.asm` includes the “`MakeDepend.h`” and `inits.dat` files. When preprocessing with

```
pp -MM vectAdd.asm
```

the preprocessor appends the `.DOJ` extension to the source file name for the list of dependencies:

```
vectAdd.doj: MakeDepend.h
vectAdd.doj: inits.dat
```

When the `-o` *`filename`* option is used with `-MM`, the preprocessor outputs the make dependencies list to `stdout`.

-Mo *filename*

The `-Mo` (output make rule) switch specifies the name of the make dependencies file that the preprocessor generates when you use the `-M` or `-MM` switch. If the named file is not in the current directory, you must provide the pathname in the double quotation marks (“”).

The `-Mo` *`filename`* option takes precedence over the `-o` *`filename`* option.

-Mt *filename*

The `-Mt` (output make rule for the named source) switch specifies the name of the source file for which the preprocessor generates the make rule when you use the `-M` or `-MM` switch. If the named file is not in the current directory, you must provide the pathname in the double quotation marks (“”).

-o [*filename*]

The `-o` (output) switch directs the preprocessor to use the specified *`filename`* argument for the preprocessed assembly file. The preprocessor uses the input file name for the output and appends an `.IS` extension if you do not use the switch or omit its argument.

-v[erbose]

The `-v` or `-verbose` (verbose) switch directs the preprocessor to output the version of the preprocessor program and information for each phase of the preprocessing.

-version

The `-version` (display version) switch directs the preprocessor to display the version information for the preprocessor program.

Preprocessor Directives/Commands

This section provides reference information about the ADSP-21xxx DSP preprocessor directives, including syntax and usage examples.

Preprocessor directive syntax must conform to the following rules:

- Must be the first nonwhite space character on its line. The exceptions to this rule are the string (`#`) and concatenate (`##`) preprocessor commands
- Cannot be more than one line in length unless the backslash character (`\`) is inserted
- Can contain comments containing the backslash character (`\`)
- Cannot come from a macro expansion

When the preprocessor evaluates an expression, the following major conditions must be satisfied:

- No invalid decimal numbers — a decimal number may be invalid if it contains characters other than the digits '0' - '9'. If the number was intended to be a hexadecimal number, precede the number with '`0x`'. Error example: `#if 123abc`.
- No invalid hexadecimal numbers — a hexadecimal number may be invalid if it is incomplete, there are no digits after the '`0x`', or if it contains characters other than the digits '0' - '9' and the letters 'a' - 'f'. Error example: `#if 0x123abct.`
- No invalid octal numbers — an octal number may be invalid if it contains characters other than the digits '0' - '7'. The leading '0' is the indicator that a number is an octal number.
  Error example: `#if 0abc`.
- No right operand of "`%`" can be zero. Error example: `#if 3%0`.

- No character constant with more than three octal digits can appear in an expression.
- A character constant must be terminated with an ending “ ‘ “.
- No bad number format may be used in an expression.
  Error example: `#if 01ux`.
- Any expression in an `#if` directive evaluates to an integer.

Table 3-3 lists the preprocessor directive set. A detailed description of each directive follows the table.

Table 3-3. Preprocessor Directive Summary

Directive	Description
`#define` (page 3-19)	Defines a macro.
`#elif` (page 3-21)	Sub-divides an `#if` … `#endif` pair.
`#else` (page 3-22)	Identifies alternative instructions within an `#if` … `#endif` pair.
`#endif` (page 3-23)	Ends an `#if` … `#endif` pair.
`#error` (page 3-24)	Reports an error message.
`#if` (page 3-25)	Begins an `#if` … `#endif` pair.
`#ifdef` (page 3-26)	Begins an `#ifdef` … `#endif` pair and tests if macro is defined.
`#ifndef` (page 3-27)	Begins an `#ifdef` … `#endif` pair and tests if macro is not defined.
`#include` (page 3-28)	Includes contents of a file.
`#line` (page 3-29)	Outputs specified line number before preprocessing.

Table 3-3. Preprocessor Directive Summary (Cont'd)

Directive	Description
`#undef` (page 3-30)	Removes macro definition.
`#warning` (page 3-31)	Reports a warning message.
`#` (page 3-32)	Converts a macro argument into a string constant.
`##` (page 3-33)	Concatenates two strings.
`?` (page 3-34)	Generates unique labels for repeated macro expansions.

#define

The `#define` command has two functions: defining symbolic constants and defining macros.

When you define a symbolic constant in your source code, the preprocessor substitutes each occurrence of the constant with the defined text or value. Defining this type of macros has the same effect as using the `Find/Replace` feature of a text editor, although it does not replace literals in the double quotation marks (“ “).

When you define a macro in your source code, the preprocessor replaces all subsequent occurrences of the macro reference with its definition. For macro definitions that are longer than one line, use the backslash character (`\`) at the end of each line except the last. End the last line with a carriage return. If you define a macro within the `do {…} while` pair, place a semicolon after the macro definition. You can add arguments to the macro definition. The arguments are separated by commas symbols that appear within parentheses.

The preprocessor has five predefined macros, `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__`. When a reserved name is used in an `#define` command, an error message is displayed. In addition, the operator "`defined`" can not appear in an `#define` command.

Syntax:

```
#define macroSymbol replacementText

#define macroSymbol[(arg1,arg2,…)] replacementText
```

where:

- *`macroSymbol`* — Macro identifying symbol.
- *`(arg1,arg2,…)`* — Optional list of arguments enclosed in parenthesis and separated by commas. No space is permitted between the macro name and the left parenthesis.

- `replacementText` — Series of instructions or a constant definition to substitute each occurrence of `macroSymbol` in your source code.

Examples:

```
#define BUFFER_SIZE 1020
      /* Defines a constant named BUFFER_SIZE and sets its
         value to 1020.
      */

#define MIN(X, Y) ((X) < (Y)? (X): (Y))
      /* Defines a macro named MIN that selects the minimum of
         two numeric arguments.
      */

#define copy(src,dest) \
r0=DM(src); \
PM(dest)=r0
      /* Defines a macro named copy with two arguments.
         The definition includes two instructions that copy a
         word from data memory to program memory.
         For example,
         copy(0x3f,0xC0);
         calls the macro, passing parameters to it.

         The preprocessor replaces the macro with the code:
         r0=DM(0x3f);
         PM(0xC0)=r0
      */
```

#elif

The `#elif` command (else if) is used within an `#if … #endif` pair. The `#elif` includes an alternative condition to test when the initial `#if` condition evaluates as `FALSE`. The preprocessor tests each `#elif` condition inside the pair and processes instructions that follow the first true `#elif`. You can have an unlimited number of `#elif` commands inside one `#if … #end` pair.

Syntax:

```
#elif condition
```

where:

- *`condition`* — Expression to evaluate as `TRUE` (non-zero) or `FALSE` (zero).

Example:

```
#if X == 1
…
#elif X == 2
…
…        /* The preprocessor executes instructions
…           following #elif when x!=1 and x=2. */
#else
…
#endif
```

#else

The `#else` command is used within an `#if … #endif` pair. It adds an alternative instruction to the `#if … #endif` pair. You can use only one `#else` command inside the pair. The preprocessor executes instructions that follow `#else` after all the preceding conditions are evaluated as `FALSE` (zero). If no `#else` text is specified, and all preceding `#if` and `#elif` conditions are `FALSE`, the preprocessor does not process any instructions inside the `#if … #endif` pair.

Syntax:

```
#else
```

Example:

```
#if X == 1
…
#elif X == 2
…
#else
…
…         /* The preprocessor executes instructions
             after #else when x!=1 and x!=2. */
#endif
```

#endif

The `#endif` command is required to terminate `#if … #endif`, `#ifdef … #endif`, and `#ifndef … #endif` pairs. Ensure that the number of `#if` commands matches the number of `#endif` commands.

Syntax:

```
#endif
```

Example:

```
#if condition
…
…
#endif
          /* The preprocessor executes instructions after #if
             when condition evaluates as TRUE. */
```

#error

The `#error` command is used to specify text that the preprocessor outputs if an error occurs during preprocessing. The preprocessor uses the text following the `#error` command as the error message.

Syntax:

```
#error messageText
```

where:

- `messageText` — User-defined text. To break a long `messageText` without changing its meaning, place the backslash character (`\`) at the end of each line except the last.

Example:

```
#ifndef __ADSP21060__
#error \
     MyError:\
     Expecting an ADSP-21060. \
     Check the Linker Description File!
#endif
```

#if

The `#if` command begins an `#if … #endif` pair. Statements inside an `#if … #endif` pair can include other preprocessor commands and conditional expressions. The preprocessor processes instructions inside the `#if … #endif` pair only when *`condition`* that follows the `#if` evaluates as `TRUE`. The preprocessor requires that an expression in a `#if` evaluates to an integer; otherwise, an error message is displayed. The number of `#if` command must equal the number of `#endif` commands.

Syntax:

```
#if condition
```

where:

- *`condition`* — Expression to evaluate as `TRUE` (non-zero) or `FALSE` (zero).

Example:

```
#if x!=100   /* test for TRUE condition */
…
…            /* The preprocessor executes instructions
                after #if only when x!=100 */
#endif
```

When a character constant (i.e., *`condition`*) in an expression uses numeric escape codes, and more than three octal digits appear in the constant, an error message is displayed. Correct the expression so it has a valid character constant (three or less octal digits).

#ifdef

The `#ifdef` (if defined) command begins an `#ifdef … #endif` pair and commands the preprocessor to test whether macro is defined. The preprocessor considers a macro defined if it has a non-zero value. The number of `#ifdef` command must match the number of `#endif` commands.

Syntax:

```
#ifdef macroSymbol
```

where:

- *`macroSymbol`* —Macro created with the `#define` command.

Example:

```
#ifdef __ADSP21060__
         /* tests for ADSP-21060 code */
#endif
```

#ifndef

The `#ifndef` command ("if not defined") begins an `#ifndef … #endif` pair and directs the preprocessor to test for an undefined macro. The preprocessor considers a macro undefined if it has no defined value or has a defined value of zero. The number of `#ifndef` commands must equal the number of `#endif` commands.

Syntax:

```
#ifndef macroSymbol
```

where:

- *`macroSymbol`* — Macro created with the `#define` command.

Example:

```
#ifndef __ADSP21060__
      /* tests for -ADSP-21060 code */
#endif
```

#include

The `#include` command directs the preprocessor to insert the text from a header file at the command location. There are two types of header files: system and user. The only difference to the preprocessor between these two types of files is way the preprocessor searches for them. The searches differ as follows:

- System Header `<fileName>` — The preprocessor searches for a system header file in the order: (1) the directories you specify and (2) the standard list of system directories.
- User Header `"fileName"` — The preprocessor searches for a user header file in the order: (1) the current directory, then in (2) the directories you specify, and finally in (3) the standard list of system directories.

The preprocessor limits the number of opened nested `#include` files to 200. Restructure your program to use fewer than 200 of the nested include files. Also, make sure there is a terminating delimiter (>) for the filename argument.

Syntax:

```
#include <fileName>  // include a system header file

#include "fileName"  // include a user header file

#include macroFileNameExpansion
     /* Include a file named through macro expansion.
        This command directs the preprocessor to expand the
        macro. The preprocessor processes the expanded text,
        which must match either <fileName> or "fileName". */
```

Example:

```
#ifdef __ADSP21060__
#include <.\21060\include\stdlib.h>
         /* tests for -ADSP-21060 code */
#endif
```

#line

The `#line` command directs the preprocessor to output the specified line and file name of an included file. Use this command for error tracking purposes. The text following the line number argument must be in the form of a filename (surrounded by the double quotation marks).

Syntax:

```
#line lineNumber “sourceFile”
```

where:

- `lineNumber` — Number of the source line that you want to output.
- `sourceFile` — Name of the source file included in double quotation marks. The preprocessor passes this parameter to the assembler. `sourceFile` can include the drive, directory, and file extension as part of the file name.

Example:

```
#line 7 “myFile.c”
```

The `#line` command is primarily used by the compiler.

#undef

The `#undef` command directs the preprocessor to undefine the macro.

The preprocessor has five predefined macros, `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__`. When a reserved name is used in an `#undef` command, an error message is displayed. In addition, the operator "`defined`" can not appear in an `#undef` command.

Syntax:

```
#undef macroSymbol
```

where:

- *`macroSymbol`* — Macro created with the `#define` command.

Example:

```
#undef BUFFER_SIZE  /* undefines a macro named BUFFER_SIZE */
```

#warning

The `#warning` command is used to specify text that the preprocessor outputs if it issues a warning. The preprocessor uses the text following `#warning` command as the warning message.

Syntax:

```
#warning messageText
```

where:

- *`messageText`* — User-defined text. To break a long *`messageText`* without changing its meaning, place the backslash character (`\`) at the end of each line except the last.

Example:

```
#ifndef __ADSP21060__
#warning \
     MyWarning: \
     Expecting an ADSP-21060. \
     Check the Linker Description File!
#endif
```

# (String)

The `#` (string) command directs the preprocessor to convert a macro argument into a string constant. The preprocessor converts an argument into a string when macro arguments are substituted into the macro definition.

The preprocessor handles white space in string-to-literal conversions according to the following rules:

- Ignores leading and trailing white spaces
- Converts any white space in the middle of the text to a single space in the resulting string

Syntax:

```
#toString
```

where:

- *`toString`* — Constant or macro definition to convert into a literal string. If the `#` operator precedes a macro parameter, the preprocessor includes a converted string into the double quotation marks (“”).

Example:

```
#define WARN_IF(EXP) \
fprintf (stderr, "Warning: " #EXP "\n")
/* Defines a macro that takes an argument and converts the
   argument to a string:

WARN_IF(current < minimum);
   Invokes the macro passing the condition.

fprintf (stderr, "Warning: " "current < minimum" "\n");
   Note that the #EXP has been changed to current < minimum and
   is enclosed in “”
*/
```

## (Concatenate)

The `##` (concatenate) command directs the preprocessor to concatenate two strings. When you define a macro, you request concatenation with `##` in the macro body. The preprocessor concatenates the syntactic tokens on either side of the concatenation operator.

Syntax:

`string1##string2`

Example:

```
/* The example code segment defines a macro that takes the name
of a command as an argument, converts the argument to a string,
and concatenates the string with _command to make the function
name.
*/

#define COMMAND(NAME) {#NAME, NAME##_command}
struct command commands[ ] =
    {
    COMMAND(quit),
    COMMAND(help),
    };
/* The code above shows the code you input to the preprocessor,
and the code below shows the preprocessor output.
*/

struct command commands[ ] =
    {
    { "quit", quit_command } ,
    { "help", help_command } ,
    };
```

? (Generate a Unique Label)

The "`?`" operator directs the preprocessor to generate unique labels for iterated macro expansions. Within the definition body of a macro (`define`), you can specify one or more identifiers with a trailing question mark (`?`) to ensure that unique label names are generated for each macro invocation.

The preprocessor affixes " `_num` " to a label symbol, where `num` is a uniquely generated number for every macro expansion. For example:

```
abcd?===>abcd_1
```

If a question mark is a part of the symbol that needs to be preserved, ensure that "`?`" is delimited from the symbol. For example:

“`abcd?`” is a generated label, while “`abcd ?`” is not.

Example:

```
#define loop(x,y)mylabel?:x =1+1;\
x =2+2;\
ourlabel?:y =3*3;\
y =5*5;\
JUMPJUMP mylabel?;\
JUMP yourlabel?;
loop (bz,kjb)
loop (lt,ss)
loop (yc,jl)

//Generates the following output:
mylabel_1:bz =1+1;bz =2+2;yourlabel_1:kjb =3*3;kjb = 5*5;JUMP
mylabel_1;JUMP yourlabel_1;

mylabel_2:lt =1+1;lt =2+2;yourlabel_2:ss =3*3;ss =5*5;
JUMP mylabel_2;

JUMP yourlabel_2;
```

```
mylabel_3:yc =1+1;yc =2+2;yourlabel_3:jl =3*3;jl =5*5;
JUMP mylabel_3;

JUMP yourlabel_3;
```