

4 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 assembler and linker from the operating system's command line or from the VisualDSP++ IDE. These tools accept options for the preprocessor on their command lines and pass them to the preprocessor. The preprocessor also can operate from the command line with its own command-line switches. For information about command-line interface, refer to [page 4-8](#); a list of preprocessor commands appears in [Table 4-3 on page 4-17](#).

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 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 assem-

Preprocessor Guide

bly source files. For more information, see the *VisualDSP++ 2.0 C Compiler and Library Manual for ADSP-218x DSPs*.

This chapter provides reference information about the preprocessor's operations. The reference information available on this topics is as follows:

- [“Preprocessor Command-Line Interface” on page 4-8](#)

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

- [“Preprocessor Commands and Operators” on page 4-17](#)

This section describes the preprocessor's commands and operators, 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++ IDE. Software developers using the preprocessor should be familiar with the following operations:

- [“Using Header Files” on page 4-3](#)
- [“Writing Macros” on page 4-4](#)
- [“Writing Compound Statements As Macros” on page 4-6](#)
- [“Using Predefined Macros” on page 4-6](#)

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 `.../include/sys` subdirectory of the VisualDSP++ installation directory.

- 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 "def2181.h"
#include "fft_ovly.h"
```

User header files are installed in the `.../include` folder subdirectory of the VisualDSP++ installation directory. This directory also includes run-time library files.

For syntax information and usage examples on the `#include` preprocessor command, see [“#include” on page 4-28](#).

Writing Macros

The preprocessor processes macros in your 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 facilitate text replacement, header file inclusion, and conditional assembly.

Macro definitions start with the `#define` and end with a carriage return. 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. 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.

Example:

```
#define false 0

#define AR_SET_T0_2 \
    AR = 0x0002;\
    L1 = 0;\
    M2 = 1;
```

A macro can have arguments. When you pass parameters to a macro, the macro serves as a template for text substitution that is usable in many different programs. The block of instructions that the preprocessor substitutes can vary with each new set of parameters. 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.

To invoke a parameterless macro, simply write the name of the macro. To invoke a macro with parameters, follow the name with parentheses surrounding the parameters. Unlike at the macro definition, there can be a space between the name and the left parenthesis.

Example:

```
#define noparm          my_variable_name
#define hasparm(x)      x + x

/* then the code: */
AR = noparm;
noparm = noparm(10); // every instance of noparm replaced
AR = hasparm;
hasparm = hasparm(10); // only instances of hasparm() replaced

/* will result in: */
AR = my_variable_name;
my_variable_name = my_variable_name(10);
AR = hasparm;
hasparm = 10 + 10;
```

For more syntax information and usage examples for the `#define` preprocessor command, see [“#define” on page 4-19](#).

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 or multi-statement macros.

You can terminate all the statements or all but the last statement of the multi-statement macro with a semicolon (;). It is a good programming practice to end the last macro statement of a macro without a semicolon and to use a trailing semicolon at the macro invocation. When ending the last statement of a macro with a semicolon, ensure that its macro invocation does not add another termination character.

Using Predefined Macros

In addition to macros you define, some DSP development tools include predefined macros that you can use in your code. Notably, the assembler preprocessor provides 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 expandable within comments.

The predefined macros that `pp` provides are listed and described in [Table 4-1](#).

Table 4-1. Predefined Macros

Macro	Definition
<code>__DATE__</code>	The preprocessor defines <code>__DATE__</code> as current date in the format 'Mm dd yyyy', for example, 'Oct 02 2000'.
<code>__FILE__</code>	The preprocessor defines <code>__FILE__</code> as the name and extension of the file in which the macro is defined, for example, 'macro.asm'.

Table 4-1. Predefined Macros (Cont'd)

Macro	Definition
<code>__LINE__</code>	The preprocessor defines <code>__LINE__</code> as the line number in the file at which the macro appears.
<code>__STDC__</code>	The preprocessor defines <code>__STDC__</code> as 1.
<code>__TIME__</code>	The preprocessor defines <code>__TIME__</code> as current time in the 24-hour format 'hh:mm:ss', for example, '06:54:35'.
ADI	The preprocessor defines ADI as 1.

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

Setting Preprocessor Options

When developing a DSP project, you may find it useful to modify the preprocessor's default options. Because the assembler and linker automatically run the preprocessor as your program is built (unless you skip the preprocessing 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 about these switches, see the [“Preprocessor Command-Line Interface” on page 4-8](#).
- From VisualDSP++, you select the preprocessor's options in the **Assemble**, **Compile**, and **Link** tabs of the **Project Options** dialog box. For more information about these option settings, see the *User's Guide for ADSP-21xx Family DSPs* and online Help.

Preprocessor Command-Line Interface

Preprocessing is the first step in the process of assembling and linking of your programs. The `pp` preprocessor also can operate independently from the command line with its own command-line switches (options). 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 linkage

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 string constant.
- The comment conversion utility replaces `!"` and `"{"` comment delimiters with C-style and C++ style comment delimiters (`/* */`, `/**/`). The program is included in your DSP development kit. For information about the comment converter, refer to [“Comment Converter” on page A-1](#).

Table 4-2 lists the `pp.exe` option set. A brief description of each option appears on [page 4-11](#).

Table 4-2. Preprocessor Command-Line Switches

Switch Name	Description
<code>-cpredef</code>	Outputs strings within <code>" "</code> .
<code>-cs!</code>	Disregards <code>!"</code> style comments.
<code>-cs/*</code>	Disregards <code>"/ * */</code> style comments.
<code>-cs//</code>	Disregards <code>"/ /</code> style comments.
<code>-cs{</code>	Disregards <code>"{ }"</code> style comments.
<code>-csall</code>	Disregards comments in all formats.
<code>-Dmacro[=definition]</code>	Defines <i>macro</i> .
<code>-h[elp]</code>	Outputs a list of command-line switches.
<code>-idirectory</code>	Searches <i>directory</i> for included files.
<code>-M</code>	Makes dependencies only, does not assemble.
<code>-MM</code>	Makes dependencies and assembles.
<code>-Mo filename</code>	Specifies <i>filename</i> for the make dependencies output file.
<code>-Mt filename</code>	Makes dependencies for the specified source file.
<code>-o filename</code>	Outputs named object file.
<code>-stringize</code>	Disables the <code>#</code> (to string) operator.

Preprocessor Guide

Table 4-2. Preprocessor Command-Line Switches (Cont'd)

<code>-v[erbose]</code>	Displays information about each pp phase.
<code>-version</code>	Displays version info about the preprocessor program.

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 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.

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 preprocessing phase.
- o bin\p1.is — Specifies the name, type, and location for the intermediate preprocessed file.

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

A summary of the preprocessor's command-line switches appears in [Table 4-2 on page 4-9](#). A brief description of each switch appears in the following sections.

Preprocessor Switch Description

-cpredef

The `-cpredef` (C-style definitions) switch directs the preprocessor to output strings enclosed within “double quotation marks”, which are compatible with C-style strings.

-cs!

The `-cs!` switch directs the preprocessor to pass the `!"` single-line comment format. The preprocessor omits the first encountered exclamation point, even if it does not denote a comment or is a part of a string, along with all following characters till the end of the current line.

The assembler accepts and processes `!"` as a comment delimiter except within preprocessor directives. This allows assembly programs to use `!"` as a comment and still support the `!="` operator in `#if` statements. A consequence is that a `!"` in a `#define` statement is not be treated as a comment. To avoid unintended effects in your programs, always use the `//` comment format.

Here are two code examples, one with the `//"` comment and one with `!"` comment formats, that demonstrate the possible problem:

```
#define x 10          // some comment explaining why x is 10
y = x + x;
```

Preprocessor Guide

gets translated into:

```
y = 10 + 10;
```

because the `//` comment is removed.

```
#define x 10      ! number of fingers  
y = x + x;
```

will result in:

```
y = 10 ! number of fingers + 10 ! number of fingers ;
```

since the assembler still recognizes `!"` as a comment, the result is:

```
y = 10
```

`-cs/*`

The `-cs/*` switch directs the preprocessor to pass the `"/ * */` multi-line comment format. The preprocessor omits the first encountered `"/ *` character combination, even if it does not denote a comment or is a part of a string, along with all following characters until the terminating `*/`.

`-cs//`

The `-cs//` switch directs the preprocessor to pass the `"/ /"` single-line comment format. The preprocessor omits the first encountered pair of slashes, even if it does not denote a comment or is a part of a string, along with all following characters till the end of the current line.

`-cs{`

The `-cs{` switch directs the preprocessor to pass the `"{ }"` single-line comment format. The preprocessor omits the first encountered `"{"`, even if it does not denote a comment or is a part of a string, along with all following characters until the terminating `"}"`.

-csall

The `-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';
                  // note single quotes for the string
                  // as the assembler preferred style.
```

-h[elp]

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

-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` and the pathname.

Preprocessor Guide

The preprocessor searches for included files in the following order:

1. current project (`.dpj`) directory
2. `\include` subdirectory of the VisualDSP++ installation directory
3. specified directory or a list of directories (in the list order)



Current directory is your `*.dpj` project directory, not the directory of the assembler program. Usage of full pathnames for the `-I` switch on the command line (omitting the disk partition) is recommended. For example,

```
pp -I "\bin\include\buffer1.dat"
```

-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 (`"`).

Preprocessor Guide

-o [*filename*]

The `-o` (output) switch directs the preprocessor to output an intermediate preprocessed assembly file, *source_name.is*, where *source_name* is the name of the source file being processed. This is the default mode. When the `-o` option is omitted entirely, no file is created, and the preprocessor output is sent to `stdout`.

The `-o` switch argument *filename.is* overrides the default option and directs the preprocessor to use the specified filename and file extension for the preprocessed assembly file.

-stringize

The `-stringize` switch directs the preprocessor to turn off the `#` (to string) operator and to enable the standard ANSI C stringize operator, thus enabling boolean constant definitions.

The `-stringize` switch is the default mode for the ADSP-21xx assemblers.

-v[erbose]

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

-version

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

Preprocessor Commands and Operators

This section provides reference information about the ADSP-218x family DSPs preprocessor commands, including syntax and usage examples.

Preprocessor command 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 operators
- Cannot be more than one line in length unless the backslash character (`\`) is inserted
- Cannot come from a macro expansion

[Table 4-3](#) lists the preprocessor command set. A detailed description of each command follows the table.

Table 4-3. Preprocessor Commands

Command/Operator	Description
<code>#define</code> (page 4-19)	Defines a macro.
<code>#elif</code> (page 4-21)	Sub-divides an <code>#if ... #endif</code> pair.
<code>#else</code> (page 4-22)	Identifies alternative instructions within an <code>#if ... #endif</code> pair.
<code>#endif</code> (page 4-23)	Ends an <code>#if ... #endif</code> pair.
<code>#error</code> (page 4-24)	Reports an error message.
<code>#if</code> (page 4-25)	Begins an <code>#if ... #endif</code> pair.
<code>#ifdef</code> (page 4-26)	Begins an <code>#ifdef ... #endif</code> pair and tests if macro is defined.

Preprocessor Guide

Table 4-3. Preprocessor Commands (Cont'd)

Command/Operator	Description
<code>#ifndef</code> (page 4-27)	Begins an <code>#ifdef ... #endif</code> pair and tests if macro is defined.
<code>#include</code> (page 4-28)	Includes contents of a file.
<code>#line</code> (page 4-29)	Outputs specified line number before preprocessing.
<code>#undef</code> (page 4-30)	Removes macro definition.
<code>#warning</code> (page 4-30)	Reports a warning message.
<code>#</code> (page 4-31)	Converts a macro argument into a string constant.
<code>##</code> (page 4-32)	Concatenates two strings.
<code>?</code> (page 4-33)	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 multi-statement macro definitions, terminate all the statements or all but the last statement with a semicolon (;). It is a good programming practice to end the last macro statement without a semicolon and to use a trailing semicolon at the macro invocation. When ending a statement macro with a semicolon, ensure that its macro invocation does not add another termination semicolon.

You can add arguments to the macro definition. The arguments are separated by commas symbols that appear within parentheses.

Syntax:

```
#define macroSymbol replacementText

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

Where:

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

Preprocessor Guide

- *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 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
```

Preprocessor Guide

#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. */
```

Preprocessor Guide

#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 __ADSP2181__
#error \
    MyError:\
    Expecting an ADSP-2181. \
    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 number of `#if` commands must equal the number of `#endif` commands.

Syntax:

```
#if condition
```

Where:

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

The conditional expression can include the `define()` operator, as in:

```
#if defined (__ADSP2181__)
```

which has the same meaning as:

```
#ifdef __ADSP2181__.
```

The advantage of the first form is that it can include multiple conditional operands as in:

```
#if defined(__ADSP2181__) || defined (__ADSP2183__).
```

Example:

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

#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.

The conditional expression can include the `define()` operator, as in:

```
#if defined (__ADSP2181__)
```

which has the same meaning as:

```
#ifdef __ADSP2181__.
```

The advantage of the first form is that it can include multiple conditional operands as in:

```
#if defined(__ADSP2181__) || defined (__ADSP2183__).
```

Example:

```
#ifdef __ADSP2181__  
    /* tests for ADSP-2181 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 __ADSP2181__  
/* tests for ADSP-2181 code */  
#endif
```

Preprocessor Guide

#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, (2) the directories you specify, and finally, (3) the standard list of system directories.

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 __ADSP2181__
#include <.\include\stdlib.h>
    /* tests for ADSP-2181 code */
#endif
```

#line

The `#line` command tells the preprocessor that the next line of the source comes from the particular line and file. The preprocessor generates error messages based on the `#line` information and passes this information to the assembler and linker through the preprocessor output. Use this command for error tracking purposes.

Syntax:

```
#line lineNumber "sourceFile.ext"
```

Where:

- *lineNumber* — Number of the source line.
- *sourceFile* — Name of the source file included in the double quotation marks. The preprocessor passes this parameter to the assembler and linker. *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.

Preprocessor Guide

#undef

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

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 __ADSP2181__  
#warning \  
    MyWarning: \  
    Expecting an ADSP-2181. \  
    Check the Linker Description File!  
#endif
```

(String)

The # (string) operator converts a macro parameter into a string constant. The preprocessor surrounds the macro parameter with string delimiters ('single quotation marks').

Syntax:

#toString

Where:

- *toString* is a macro parameter. If the # operator appears in a macro definition, it must be followed by a macro parameter.

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 ', the preferred assembler style for
   strings.
*/
```

(Concatenate)

The ## (concatenate) operator concatenates two strings. When you define a macro, you request concatenation with ## in the macro definition. 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`), a programmer 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; \
yourlabel?: 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;
```

