C_PREP - Documentation Version 1.9B

by Jim McDowell
JML SOFTWARE DESIGN

With 1.9 updates by
Gene Heskett, CE@WDTV, delphi WOAY
And 1.9A,B updates by
Willard Goosey, goosey@sdc.org

C_PREp 1.9B fixes two more errors: Arrays of structs and struct expressions now would break element names if the element name contained an '_', and the Microware /d string escape is no longer flagged as an error.

C_PREP 1.9A has three more bugs fixed: __FILE__, __DATE__, and __TIME__ macros now output double-quote enclosed strings like the ANSI standard and this file says they should, a Y2K bug in __DATE__ was worked around (the real fix would go in clib), and __STDC__ now has the proper value of 1 in ANSI mode.

C_PREP Version 1.9 has had two bugs fixed, and one error printout re-enabled that the 1.8 version didn't do, and its array sizes for defines increased from a max of 300 and 5k of string space for the defines to 400 and 6500 bytes of string space for the names.

The two bugs were:
1, An erronious asterisk in front of the variable named lpcntr in the function "expand(ln, loop, lpcntr)" located in the file "cp4.c" which caused it to check the value lpcntr pointed at instead of the value of lpcntr. In the case of the first call as opposed to a recursion call, that was NULL.

Unforch, the data at *NULL wasn't always NULL too. In the event *NULL wasn't NULL too, it wrote all over the pathlist array in the direct page, rather effectivly crashing the program because it couldn't close its paths and exit properly..

And 2, The "#undef"ine function didn't work due to how the function splittok() worked compared to the way the names were stored in the string array for defined names. In the array, it was 'HOWMANY \0". It was passed an isolated 'HOWMANY\0' name, no spaces on either end, which was the subjected to the splittok() function which added a space on both ends of the name, or ' HOWMANY \0' before doing the name comparisons. The compare failed of course because of the leading space which is not stored in the name define string array, therefore the undef was never found and done. Each defined name does have a space on the end of it however, presumably so its spaced properly in the output when the define is substituted by this program. The cure was to inc the passed line pointer one byte before doing the compares thereby skipping the leading space as far as strcmp() was concerned. I also added a return(line[0]=\0) to the end of that function so that it didn't waste the time looking thru the rest of a 350 name long list once it had found that name and undefined it by removing it and its associated data from the define arrays. Thats no real speedup as it would only count for the #undef line, one out of maybe 120 pages of source code.

I also re-enabled an additional error printout, showing a karat under the first character of an error after the line has been printed, after the nature of the error has been reported. Thats not fully checked out and may not point at the first character of the offending error under some conditions.

These two errors are the only ones I'm aware of but that doesn't mean there won't be more found. In the event you become aware of another problem, please feel free to advise me of it by leaving email to WOAY here on Delphi.

For those who would use this to recompile sz3_24 again, there is a 4 line

section of code 125 lines into the sz.c file that used to be ignored and
reported as an error by c_prep18. It undefs HOWMANY and redefines it. I'm
not sure what should be done with those 4 lines of code, Paul Jerkatis
(MITHELEN) is our resident expert on that ust yet, and the question has
been put to him. I've simply commented it out in my source, but then I don't
expect my particular copy of that source will ever be compiled on a POSI
system for instance.

After this paragraph, the rest of this is Jim McDowells very well organized
and thought out effort. I couldn't have found the errors that were made so
easily had this code not been so concise and error free as issued by Jim
when he uploaded this as the last effort he was putting into the OS9
programmers kit. Many Thanks to you Jim McDowell.
Gene Heskett, WOAY on Delphi - end of added notes

C_PREP is an ANSI standard C preprocessor. It can completely replace
Microware's c.prep. It contains all of the features of c.prep plus some
extras. It follows the grammar set out in the ANSI C standard by the   3J11
committee.

Version 1.8 runs 1.4 times the speed of c.prep. This should be an acceptable
speed tradeoff for the extra flexability that C_PREP offers. Preprocessing
is a small part of the compiling process. C_PREP took 55 seconds to process
a 16K source file. C.PREP took 38 seconds.

If your program uses complex   define macros you may get a STACK OVERFLOW
error. If this happens ust specify extra memory on the command line (32k).
I STILL WOULD APPRECIATE INPUT ON HOW THIS PROGRAM OPERATES. IF THERE ARE
ANY BUGS PLEASE REPORT THEM TO ME SO I CAN FI   THEM.

C_PREP COMMANDS

ersion 1.8 supports the following preprocessor commands:

(races   enclose re uired information, rackets [] optional information.)

```
#asm                /* If in MicroWare mode */
#endasm             /* else it uses #pragma */
#define  identifier    token se  uence
#define  identifier (  identifier-list  )  token se  uence
#undef  identifier           /* Nope, not till 1.9 folks */
#include     filename
#include "  filename  "
#include  token-se  uence
#if  constant-expression
#ifdef  identifier
#ifndef  identifier
#elif  constant-expression
#else
#endif
#error  token-se  uence
#line  constant  ["  filename  "]
#line  token-se  uence
#pragma  token-se  uence
#
defined [(]  identifier  [)]
```

ersion 1.8 also supports the predefined names:

```
__LINE__              (  eturns current line number)
__FILE__              (  eturns current file name)
__DATE__              (  eturns compile date: Mmm dd yyyy)
__TIME__              (  eturns compile time: hh:mm:ss)
__  TD  __            (  eturns 1 if in AN  I mode  otherwise it is
                         undefined)
```

This version also has the following AN I compatible features:

* Optional trigraph expansion (engaged with -t switch)
*   oncatenation of any line ending with a \ and newline character
* Lines are split into tokens separated by spaces
* Each comment is replaced by a single space
* Handles macro usage of # (places " around following argument)
* Handles macro usage of ## (concatenates preceding and following tokens)
*   oncatenates adjactent string literals

The following features are included also:

* When c_prep is run as a filter the root source file is given a default name of "stdin.c"
* In the AN I (-c) mode, instead of #asm and #endasm the following pragma will be generated: #pragma asm   assembly code
* Microware c.prep emulation:
  - supports Microware's oddball line codes (default)
  - supports -e switch to set edition number
  - supports -l switch to copy source code to the compiler for including comments in the assembly language output.
  - allows #asm and #endasm to include assembly code in the   source
* The e and l switches generate the following pragma commands:

```
#pragma edn   edition #
#pragma src      source line
```

* Error handling. The filename and line number are returned. The line number counts from 0-... (compatible with sled). The source line is reprinted to aid in debugging.   ometimes the source line printed will not be the same as in the source file. This is because the preprocessor alters the line as it goes.   o when an error is encountered the line printed with the error code may reflect preprocessor changes.

 ersion 1.8 has the following limitations:

* Maximum line length is 250 characters. This is less than the AN I

recommendation but cuts in half the needed memory. It should suit most programmer's needs, but can be changed by changing the variable LINEMA   in cp.h and then recompiling.

Maximum memory allocated for definition data is 6500 bytes. This can be changed by STRNG_TBL_MA   in cp.h.

Maximum number of arguments allowed in a macro (define) is 4. The maximum length of each argument is 120 characters. This is less than the ANSI recommmendation but should suit most needs. These limits can be ad usted by changing MA   _ARGS and MA   _LENGTH in cp.h.

Maximum number of nested   if statements is 8. This is the ANSI standard. It can be changed by MA   _NEST_IF in cp.h.

Maximum number of nested   include statements is 8 levels beyond the root file. This can be changed by MA   _INCLUDE in cp.h.

Maximum number of definitions allowed is 400. This can be changed by MA   _DEFS in cp.h.

  include   statements look for files in /DD/DEFS directory.

C_PREP's usage syntax is as follows:

```
   c_prep    -opt1     -opt2     ...       filename       -opt1     -opt2     ...
```

The program outputs to the standard output path. It can be run as a filter:

```
   list file.c   c_prep
   list test.c   c_prep   test.prep
```

or it can read the input file from the command line:

```
   c_prep file.c
```

other options include:

```
   c_prep -h  (will produce help screen)
```

```
    c_prep -t test.c   (will perform trigraph substitution step)
    c_prep -dMAIN test.c   (will define MAIN as 1 before preprocessing)
    c_prep -dTWO=1   1 test.c   (will define TWO as 1   1 before preprocessing)
    c_prep file.c -c   (process file with AN  I line codes)
    c_prep -c file.c -e=5   (process with AN  I line codes     set edition
                            e  ual to 5)
    c_prep -c file.c -l   (process with AN  I line codes     copy source lines to
                          output for inclusion in the assembly listing)
```

```
                    ********************
                  ***                 ***
               ******    OMMAND   UMMA   Y  ******
                  ***                 ***
                    ********************
```

All preprocessor commands are prefixed with a # character.   _   E   allows the # to be preceded and followed by whitespace characters. Thus the following is valid:

```
          #       ifdef       MAIN
```

The following is a detailed description of the new preprocessor commands found in   _   E   ersion 1.8:
(races   enclose re uired information,   rackets [] optional information.)

#define   identifier (identifier-list   )   token se uence

```
     Example: #define dprint(expr) printf(#expr " =     g\n",expr)
              dprint(x/y)

       esult:  printf("x/y" " =     g\n",x/y)
```

Notice the phrase #expr in the definition. This tells the
preprocessor to place   uotation marks (") around expr.
In actual processing the two string literals in the result would
be concatenated.

Example: #define paste(front,back) front ## back
    paste(get,char())

  esult : getchar()

This absurd example shows the function of the ## operator. It
concatenates the preceding and following token.

#undef   identifier

Example: #undef   Y

  esult : If   Y   has been defined its definition is erased, otherwise
    the commands is ignored.

This is the ONLY allowable way to change a defined value. It must be
undefined and then redefined with a new value.

#include   filename

This command searches the LI   directory on the default drive for
the inclusion file.

#include "   filename   "

This command searches the current data directory for the inclusion
file.

include   token-sequence

```
Example:   define CTYP    ctype.h
           include CTYP
```

This command after macro expansion must be in one of the above two forms.

if   constant-expression
elif   constant-expression

```
Example:   if SYS     1
             puts("SYS  1")
           elif SYS     2
             puts("SYS  2")
           endif
```

This command evaluates the constant-expression if it is TRUE then the following lines are processed until an   elif (else-if),   else, or   endif is encountered. The constant-expression must always evaluate to TRUE (non-zero) or FALSE (zero).

The constant-expression may contain any unsigned integer or token-sequence that macro expands into an unsigned integer. Parentheses are allowed and the following operators are allowed:

Unary operators:

```
      -
```

Binary operators:

```
   /      -
```

error   token-sequence

Example: error File Error

Result : User error ... File Error

This command can be used to flag portions of your program. It will not stop the preprocessor but will write the warning message to the standard error output path.

line   constant   "   filename   "

Example: line 0 "stdin.c"

This command is supplied by the preprocessor to allow the compiler to remember the original source line numbers.
If running C_PREP in Microware emulation mode the Microware format of this command will be used instead.
You can use this command to override the compiler settings.

line   token-sequence

Example:
```
define LINE 256
define FILE "myname.c"
line LINE FILE
```

This is a variation on the above command. In this instance the arguments are expanded to produce a   line command of the format previously shown. This form is never produced by the compiler.
It is only for the programmer's convenience.

pragma   token-sequence

Example:
```
pragma asm    assembly code
pragma edn    edition number
pragma src    source code
```

The pragma commands is designed to allow implementation defined commands.

ASM replaces the Microware asm/ endasm commands. Just prefix each assembly code line with " pragma asm".

EDN and SRC are generated by CPREP automatically. EDN is controlled by the -e switch. SRC is controlled by the -l switch.

Example:

This is the NULL directive. It is ignored by the preprocessor.

defined (identifier)

Example: if defined(SYS)
    if defined SYS

The defined command tests the identifier to see if it has been defined. If it has it returns TRUE, otherwise FALSE. Parentheses are optional. In the second example the test returns TRUE if SYS is not defined.

__LINE__

Example: printf("Current line number d n",__LINE__)

Result : Current line number 450

This is a decimal constant containing the current line number.

__FILE__

Example: printf("Current file name is: s n",__FILE__)

Result : Current file name is: solve.c

This is a string literal containing the name of the file being compiled.

__DATE__

Example: printf("Compile date is: s n",__DATE__)

Result : Compile date is: Jan 25 1993

This is a string literal containing the date of compilation. It is in the form: Mmm dd yyy

__TIME__

Example: printf("Compile time: s n",__TIME__)

Result : Compile time: 12:59:59

This is a string literal containing the time of compilation. It is in the form of: hh:mm:ss

__STDC__ (Indicates whether compiler is ANSI compatible)

```
Example:   if __STDC__      1
              puts("This is an ANSI preprocessor")
            else
              puts("This is not ANSI   ")
```

__STDC__ will return a constant 1 if in ANSI mode. Otherwise it will leave __STDC__ undefined.