

New Technical Notes

Macintosh



®

Developer Support

MPW C Q&As

Platforms & Tools

M.PT.MPWC.Q&As

Revised by: Developer Support Center

October 1992

Written by: Developer Support Center

October 1990

This Technical Note contains a collection of Q&As relating to a specific topic—questions you've sent the Developer Support Center (DSC) along with answers from the DSC engineers. While DSC engineers have checked the Q&A content for accuracy, the Q&A Technical Notes don't have the editing and organization of other Technical Notes. The Q&A function is to get new technical information and updates to you quickly, saving the polish for when the information migrates into reference manuals.

Q&As are now included with Technical Notes to make access to technical updates easier for you. If you have comments or suggestions about Q&A content or distribution, please let us know by sending an AppleLink to DEVFEEDBACK. Apple Partners may send technical questions about Q&A content to DEVSUPPORT for resolution.

|New Q&As and Q&As revised this month are marked with a bar in the side margin.

Use File Manager SetEOF calls to do "lseek" file sizing

Written: 6/14/91

Last reviewed: 8/1/92

Using standard C I/O calls, I would create a new file of a specific size with the following code:

```
int n;  
n = creat( "filename" );  
lseek( n, the_size_I_want, 2);
```

However, under MPW 3.1 the lseek call always returns -1 with errno set to 6 (ENXIO). The only workaround I have found is to actually write the number of dummy bytes into the file before I have to seek around in it. Is there a call resembling the UNIX call ftruncate, or must I use stdio calls rather than fctnl calls?

MPW C lseek is not capable of increasing file sizes, as in the UNIX lseek code. The best way to achieve file sizing is to use the File Manager SetEOF calls, which can grow a file to whatever size (using Create, Open and SetEOF). Many of the low-level C I/O routines don't work exactly like they work under other platforms, and this is a known worry when moving source code between PC, Macintosh and UNIX platforms. If possible, use the ANSI C file I/O routines.

MPW C error 426 and workarounds

Written: 7/13/90
Last reviewed: 8/1/92

What does the following MPW C error mean?

```
# C - Fatal Error : 426
#-----
File "HD80:MPW:Gui:Gui:source:debug.c"; Line 2383
#-----
# C - Aborted !
```

“File "HD80:MPW:Gui:Gui:source:debug.c"; Line 2383” is a valid MPW compound command. If you triple click on it and press enter, the line is executed and it will take you to the exact line of your source code that produced the error 426.

The error 426 is an internal compiler error that occurs for several reasons, but all are related to symbol table space for local variables overflowing memory. Possible workarounds are:

- Use a bigger memory partition for MPW.
- Use less or shorter local variable names.
- Restructure your compile and linking so that less total local symbols are used in each compile.

Building resources with MPW C

Written: 12/11/90
Last reviewed: 8/1/92

When building a 'CODE' resource with MPW C the main entry point needs to be specified with the -m option of Link. So why does one need also to make sure that the routine is the first one in the file and that the file is the first one linked? Why can't “-m” be enough to tell the linker where the entry point will be?

From your question, it sounds like you are asking about resources other than 'CODE' resources containing object code (i.e., 'XMCD's, 'INIT's, 'cdev's, 'CDEF's, 'LDEF's, etc.). When you create 'CODE' resources, you are building a full-fledged application and don't need to specify the -m option or make sure that the main entry point is the first one in the link list. With applications, that is done for you.

You need to specify the -m option and make sure that the main entry point is the first one in the link list only when you build stand-alone code resources. There are separate reasons for each of these requirements.

You specify the -m option to tell the linker what the root module or routine is. It uses this module as the start of a tree search for unreferenced modules. These unreachable modules are not included in the link, making your code smaller.

The reason for making sure that the main entry point is the first one in the link list is so that clients know where the starting point of the code is. With standalone code, you don't get a jump table, which is normally the mechanism used to find the main entry point (the main entry point is in the first jump table entry). Without a jump table, you don't have an indication of “the main entry point is \$xxxx bytes into the resource,” so by convention, it's

the first byte. I think that the exception to this are 'DRVr's, which have a 5 entry table at the start of the resource that points to 5 routines in the 'DRVr'.

For more information on this, you might want to take a look at the DTS Macintosh Tech Note "Stand-Alone Code, *ad nauseam*."

Converting parameter sizes for MPW and A/UX C

Written: 3/11/91

Last reviewed: 8/1/92

Is it possible to tell the MPW C compiler to generate 68881 instructions without having the compiler represent extended numbers in 96 bits?

—

I know of no way to induce either MPW or A/UX C to change the size of the parameters it passes, but there are SANE routines to convert back and forth. MPW Sane.h conditionally (depending on whether you are using mc68881 or not) defines the types `extended80` and `extended96` as well as routines to convert between them:

```
#ifdef mc68881
struct extended80 {
    short w[5];
};
void x96tox80(const extended *x, extended80 *x80); /* x is src, x80 is dest */
void x80tox96(const extended80 *x80, extended *x); /* x80 is src, x is dest */

typedef struct extended80 extended80;
#else

typedef struct {short w[6];} extended96;
void x96tox80(const extended96 *x96, extended *x); /* x96 is src, x is dest */
void x80tox96(const extended *x, extended96 *x96); /* x is src, x96 is dest */
#endif
```

Build these routines into your MPW code, then call them from your A/UX code to do the conversion as necessary.

If you want to build these routines in assembler under A/UX, here is a dump of the code from MPW:

```
Module:          Flags=$08=(Extern Code)  Module="x96tox80" (313)
Segment="CSANELib" (259)

Content:          Flags $08
Contents offset $0000 size $0018
00000000: 206F 0004      ' o..'          MOVEA.L    $0004 (A7),A0
00000004: 226F 0008      '"o..'         MOVEA.L    $0008 (A7),A1
00000008: 3290           '2.'          MOVE.W     (A0), (A1)
0000000A: 2368 0004 0002 '#h....'      MOVE.L     $0004 (A0), $0002 (A1)
00000010: 2368 0008 0006 '#h....'      MOVE.L     $0008 (A0), $0006 (A1)
00000016: 4E75           'Nu'          RTS
```

```
Size:             Flags $00 ModuleSize 24
```

```
Module:          Flags=$08=(Extern Code)  Module="x80tox96" (314)
Segment="CSANELib" (259)
```

```
Content:          Flags $08
Contents offset $0000 size $0018
00000000: 206F 0004      ' o..'          MOVEA.L    $0004 (A7),A0
00000004: 226F 0008      '"o..'         MOVEA.L    $0008 (A7),A1
```

00000008: 2368 0006 0008 '#h....'	MOVE.L	\$0006(A0), \$0008(A1)
0000000E: 2368 0002 0004 '#h....'	MOVE.L	\$0002(A0), \$0004(A1)
00000014: 3290	MOVE.W	(A0), (A1)
00000016: 4E75	RTS	

Size: Flags \$00 ModuleSize 24

Disable MPW dead-code stripping with #pragma directive

Written: 6/5/91

Last reviewed: 8/1/92

In my runtime MPW C object messaging mechanism, when I resolve references to routines at runtime through a little piece of code, the linker never sees any references to my routines and thinks that they are all dead code, dutifully eliminating all them during the linking process. Is there a way to force a routine to be loaded into a code resource without adding either an unnecessary reference in the jump table or its code segment?

—

MPW C 3.2 has a #pragma directive to disable dead-code stripping. It is used as follows:

```
#pragma force_active on                // don't strip the below code
void not_accessed()
{
}
#pragma force_active off                // go back to stripping as usual
```

The default for force_active is off, and so dead code will normally be stripped.

The #pragma directive didn't exist before MPW 3.2, but it is still possible to avoid code stripping with a technique similar to the one used by MacApp. Simply insert a conditional that is never true into a section of your program that _is_ called and make a function call from this conditional to the nonreferenced code, as shown below:

```
short i=0;
if (i==1)
    DeadFuncCall()
```

You can easily check to make sure your code did not get stripped by adding the MPW linker option “-uf <file>.” This option will write the names of all unreferenced modules to the specified file.

Use type “long” & “short” instead of “int” for Mac toolbox calls

Written: 6/11/91

Last reviewed: 8/1/92

Using the “int” type isn't a good idea for Macintosh toolbox calls, because the size of the int type is compiler dependent. MPW has 4-byte ints and Think C has 2-byte ints, for example. The code will compile slightly differently depending on the availability of prototypes and the compiler used. It's better to use types “long” and “short” because they are 4 bytes and 2 bytes, respectively, for all Macintosh C compilers.

Maximum buffer size for MPW C setvbuf is 65536

Written: 8/13/91

Last reviewed: 8/1/92

MPW C's `setvbuf` size parameter is treated internally as an unsigned short. This means that the largest value acceptable to `setvbuf` for its size parameter is 65536. Larger values will be treated modulo this number. In practice, the optimal settings for this value using MPW C is somewhere in the range of 8K–12K, so DTS recommends that you use a value of 10K. Alternatively you can experiment to see if other values provide better results for you. But any value you try should be within 64K. Your mileage using `setvbuf` may differ for other development environments.

MPW C `comp` keyword

Written: 12/16/91

Last reviewed: 8/1/92

What is the keyword “`comp`”? If `comp` is a reserved word in MPW C, is it also reserved for type names? If it is, then the MPW headfile `Script.h` is in violation of this at line 788:

```
typedef comp LongDateTime;
```

Where does this `comp` come from? Is this why ‘`comp`’ is reserved?

“`Comp`” is not an ANSI or C++ keyword; however, it's an IEEE-defined type to be used mainly for portable data, where floating point values are stored and retrieved between various systems/architectures (the name is short for “compatible”). By contrast, the “extended” type should be used for Macintosh-only type applications, or internally inside computation algorithms on a Macintosh system (`comp` is less precise than extended, only 64 bits).

Loading and dumping symbol files for MPW C

Written: 12/17/91

Last reviewed: 8/1/92

Loading and dumping symbol files for MPW C, I get errors that say “### Error 542 symbol table structure sensitive compile time variables in dump file do not match current settings.” I'm using two files: `Makefile`, which sets `CREATE_THE_DUMPFILE` for creating a bogus “`include.h.o`” for `Mak`; and `include.h`, which gets `#include'd` into every source file. In `Makefile`, `{COptions}` has “`-d USE_THE_DUMPFILE`” so all the source files will load `includes.h` and then load the symbol table. What could be causing this error?

You're probably getting the error because the rule you're using to build the dumpfile begins with “`C -c {SYM}...`” The `-c` option, which suppresses an output file, makes a dump file that's only useful for other compiles with `-c`.

So, you should get rid of the `-c` from your dumpfile rule; of course you can always add a line to the rule to delete the resulting `include.h.o`.

Script for inline assembly code with MPW C compiler

Written: 1/8/92

Last reviewed: 8/1/92

How can I get the MPW C compiler to do inline assembly code, like I do with Think C?

With MPW C, it still isn't possible to write inline assembly language code (at least not easily). You can write the hexadecimal equivalent of your assembled code and put that in your code (much as Apple does in many of its MPW C header files), but this is not nearly as nice as Symantec's implementation.

In MPW C, you have to type the hexadecimal code instead of the assembly statements. An example from "aliases.h" follows:

```
pascal OSErr NewAlias(const FSSpec *fromFile,
                    const FSSpec *target,
                    AliasHandle *alias)
    = {0x7002, 0xA823};
```

Fortunately there is an MPW script that helps with the task of calculating the hexadecimal equivalent to assembly language commands. The script, called A2Hex, writes hex equivalent of selected lines of assembly to standard output. The script follows below. Basically put the A2Hex file in the MPW Scripts folder, and A2Hex can be added to your Tools menu (or a different menu) with the following command:

```
AddMenu Tools A2Hex 'A2Hex "{Active}" ΣΣ "{Worksheet}"'
```

Then you can enter assembler source code. For example, if you enter this in your work sheet:

```
MoveQ #$0A, D0
LSL.L    #$8, D0
```

select it and then chose A2Hex from the menu, this is the output:

```
700A          #MOVEQ      #$0A,D0
E188          #LSL.L      #$8,D0
```

Here is the script:

```
#-----
#   A2Hex
#   MPW Shell Script
#
#   Original by Stuart Davidson
#   Changes by Gina Cherry • August 16, 1991
#   Copyright:   © 1991 by Apple Computer, Inc., all rights reserved.
#
#   Usage:      A2Hex [file]
#
#   Function:
#       A2Hex takes one or more lines of assembly language and computes the
#       equivalent in hexadecimal. The assembly lines cannot contain
#       variables, labels, or procedure headers. If an input file is not
#       specified, A2Hex prompts the user for a line of assembly language.
#       If an input file is specified, A2Hex converts the selected text in
#       the input file to hex.
#
#   Note:
#       A2Hex can be added to the Tools menu with the following command:
#           AddMenu Tools A2Hex 'A2Hex "{Active}" ΣΣ "{Worksheet}"'
#-----
#   The first section of this script consists of commands which are
#   executed by the MPW Shell. These commands are ignored by StreamEdit,
```



```
# because of the semi-colon in the first column.
#-----
# Shell Commands
#-----
; # If more than one parameter was given, write error msg & exit script
;   If {#} > 1
;       Echo "### Usage: {0} [file]" >> Dev:StdErr
;       Exit 1
;   End
;
; # Don't exit on error.
;   Set Exit 0
;
; # Write procedure header to temporary file.
;   Echo 'FOO      PROC' > "{0}Temp.a"
;
; # If an input file was specified:
;   If {#} == 1
;       # Copy selected text from input file to clipboard.
;       Copy $ "{1}" > Dev:Null
;       # Exit if no text is selected in the input file.
;       If {Status} != 0
;           Delete {0}Temp.a
;           Exit 0
;       End
;       # Save value of {NewWindowRect}
;       Set OldWindowRect "{NewWindowRect}"
;       # Want to open a small workspace.
;       Set NewWindowRect 0,0,100,100
;       # Open temporary file.
;       Open {0}Temp.a
;       # Restore old value of {NewWindowRect}
;       Set NewWindowRect "{OldWindowRect}"
;       # Append text from clipboard to temporary file.
;       Paste ∞ {0}Temp.a
;       # Write a newline to temporary file.
;       Echo >> {0}Temp.a
;       # Insert tabs.
;       # Set count to the number of lines in the temporary file.
;       Set count `Count -1 {0}Temp.a`
;       # Loop through lines.
;       Loop
;       # Break if on first line.
;       Break if {count} == 1
;       # Position cursor at beginning of current line.
;       Find Δ{count} {0}Temp.a
;       # Insert tab at beginning of line.
;       Replace /(≈)1/ 0t1 {0}Temp.a
;       # Decrement count.
;       Set count `evaluate {count} -1`
;       End
;
;   # Close temporary file
;   Close -y {0}Temp.a
;
; # If no input file was specified:
;   Else
;       # Request a line of assembly code.
;       Set assembly "`Request "Assembly to hex:"`"
;       # If Cancel button was chosen, delete temp file & exit script.
;       If "{assembly}" == ""
;           Delete {0}Temp.a
;           Exit 0
;       End
;       # Write assembly code to temporary file.
```

```
;      Echo -n %t >> {0}Temp.a
;      Echo "{assembly}" >> {0}Temp.a
;      End
;
;      Echo "%tRTS" >> {0}Temp.a
;      Echo "%tENDPROC" >> {0}Temp.a
;      Echo "%tEND" >> {0}Temp.a
;
;      # Assemble temporary file.
;      Asm {0}Temp.a
;
;      # If assembly fails, exit script.
;      If {Status} != 0
;          Echo "### {0}: Assembly failed." >> StdErr
;          Exit 3
;      End
;
;      # Pipe output of Dumpobj to StreamEdit command. Dumpobj writes
;      # disassembled code to standard output. StreamEdit executes the
;      # StreamEdit statements at the bottom of this file to extract the
;      # hex equivalent of the assembly input from the Dumpobj output. The
;      # call to StreamEdit uses the command Which "{0}", which expands
;      # into the name of the currently running shell script. The -d
;      # option causes StreamEdit to discard text from the input file,
;      # writing only the output from print statements to standard output.
;      Dumpobj -m FOO {0}Temp.a.o | StreamEdit -d -s `Which "{0}"`
;
;      # Delete temporary file(s).
;      Delete {0}Temp~
;
;      # Exit shell script.
;      Exit

#-----
-
# StreamEdit Statements
#-----
-
# These statements are meant to operate on the output of the Dumpobj
# shell command. They write each instruction (except the RTS
# instruction) in both assembly and hexadecimal to standard output.

# Find all hex lines except the one representing the RTS instruction.
(!/[0-9]<8>:~RTS~/) &&
(/[0-9]<8>: ([0-9a-fA-F ]+ )1~%~%[ ]+([~ %]+~)2/)
# Print hex and assembly equivalent to output file.
print 1'#'2
```

MPW C “offsetof” doesn’t work with array references

Written: 1/9/92

Last reviewed: 8/1/92

The MPW C “offsetof” macro from StdDef.h doesn’t seem to work for initialized globals such as the following:

```
typedef struct
{
    short bob;
    char stGolly[10];
}
FOO;
```

```
short fooArray[] =
{
    offsetof(FOO,bob),
    offsetof(FOO,stGolly)
};
```

When I run this through, the compiler reports:

```
# ((size_t)&((FOO *) 0)->stGolly)
# };
# ?
### Error 234 error in type of initialization expression
```

The offsetof(FOO,bob) worked fine, but string offsets don't compile the string offsets. Any ideas?

MPW C has a parsing bug with offsetof and arrays inside/outside the struct. Any other definitions, such as long/short..., work OK. Engineering is working on a fix for the problem.

MPW C compiler optimization

Written: 10/31/90

Last reviewed: 8/1/92

Is there a code optimizer for MPW C?

Well, the C compiler tries its best to optimize; it optimizes by default. Compared with the UNIX world (where the cc has a special -O option), in the MPW world we have a lot of special optimizing flags that the developer can freely test separately. However, we would like to point out that it is usually recommended to only test a few things/flags at a time. With the new MPW 3.2 compiler the developer is also able to switch off optimization. Try to use the performance tools in order to find out the code bottlenecks where optimization would help.

Preventing MPW C string constants from compiling as globals

Written: 10/31/90

Last reviewed: 8/1/92

Using MPW C, a local immediate text string such as printf("%s\n","This is a test") is compiled as a global variable. This is causing the global data over 32K problem. To get around this we used the "-m" option to compile some of the files that are not speed critical when running. We then link the entire program with the "-srt" option. Will this cause a degradation of performance?

The following options for the C compiler will move string constants into 'CODE' resources:

```
C [option...] [file] < file > preprocessor ≥ progress
-b      # put string constants into code & generate PC-relative references
-b2     # implies '-b' above, and allows string constants to be overlaid
-b3     # overlaid string constants in code (but not PC-relative refs.)
```

Another idea is maybe to start using 'STR ' and 'STR#' resources and load strings into memory with the Resource Manager.

MPW C compiler parameter-handling difference

Written: 10/31/90

Last reviewed: 8/1/92

In our example below, when you pass the parameter in the test procedure, the *i* value we expect is the same for the *x* and *y* array. With MPW C, the *i* for array *x* is incremented and the *i* for array *y* is not. With a different compiler, *i* is incremented after the parameter passing.

```
#include <StdIO.h>

main()
{
    double x[8], y[8];
    short i, j;

    i = j = 0;
    while (1) {
        if (i >= 8) break;
        j = i;
        test(&(x[i]), &(y[i++] ), j);    /* problem here */
        printf("i=%d x=%f y=%f\n", i, x[i - 1], y[i - 1]);
    }
}

test(x, y, i)
double *x, *y;
short i;
{
    *x = (double)i;
    *y = (double)i;
}
```

Increment/decrement of values inside function calls is not defined concerning ordering. See K&R's new edition, page 53. In general one should try to increase/decrease values before passing them as parameters. The sore point in your case is the `,` statement, which means that the compiler does not have an obligation to pass the first or second statement, or to increase the

value in a certain order. So please increment the value before passing it as a function value. Concerning optimization, there's no worry incrementing values before passing them.

MPW C 3.1 compiler -mc68020 option bug fixed for 3.2

Written: 10/31/90

Last reviewed: 8/1/92

In our example below, the bit field of record array “field1” is assigned a number 1, and then it is assigned into “field2” record array using “field1” value. When we print out the array we can consistently obtain a zero in the “field2[0].visible.” We should be getting all ones when we print the array.

```
/*
 * Here is the bug we found on MPW C compiler:
 * We compiled using -mc68020 option, and the result didn't
 * turn out to be what we expected.
 */
#include <Memory.h>

typedef struct _fldInfo {
    unsigned short visible:1, unused:15;
} ReadoutField;

/*
 * The result of this piece of code is:
 * i=0 field1=1 field2=0
 */
main()
{
    ReadoutField field1[1], field2[1];
    short i;

    for (i = 0; i < 1; i++) {
        field1[i].visible = 1;
        field2[i].visible = (field1[i].visible); /* assignment of same type */
        printf("i=%d field1=%d field2=%d\n", i, field1[i].visible,
            field2[i].visible);
    }
}
```

This was a bug with the MPW 3.1 C compiler. The latest MPW C 3.2 beta compiler on the current E.T.O. (Essentials-Tools-Objects) CD fixes this problem. Then again bitfields are a little bit nasty, especially concerning porting issues—one has to consider issues like separate padding and data structure sizing. Bit fields are supported by the language, but they tend to cause problems.

MPW C function prototyping restrictions

Written: 12/20/89

Last reviewed: 8/1/92

When is a C function prototype *not* a C function prototype?

Many developers want to take advantage of function prototyping provided by the ANSI MPW C 3.0 compiler. But the MPW C implementation is somewhat restrictive and you may be getting less than you bargained for.

Consider the following code:

```
// Declare functions, with prototypes
int foo (int a, int *b);
int bar ();
int foobar ();

// Now define foo()
int foo (a, b)
int a;
int *b;
{
    *b = a;
    return (a);
}

int bar ()
{
    int i;

    foo (1, &i);
    return (i);
}

int foobar ()
{
    return (foo (1, 2));    // The second parameter is not pointer to int
}
```

If you simply compile this code without any options, it compiles without errors, even though the call to `foo()` in `foobar()` is clearly incorrect. If you compile with the `-w2` option, the compiler issues the following warnings:

```
#
#foo (1, &i);
# ?
### Warning 270 This function has no prototype
#-----
#      File "foo.c"; Line 28
#-----
#{
#return (foo (1, 2));    // The second parameter is not pointer to int
#      ?
### Warning 270 This function has no prototype
#-----
#      File "foo.c"; Line 34
#-----
```

This may seem strange because the functions were clearly declared with prototypes.

The problem here is that in addition to using the new style function declarations, you **MUST** use new style function definitions. So the correct way to define `foo()` is:

```
int foo (int a, int *b)
{
    *b = a;
    return (a);
}
```

Now, when you compile, you correctly get the error message:

```
# {
# return (foo (1, 2));    // The second parameter is not pointer to int
#
### Error 229 mismatch between formal and actual parameter types
#-----
#      File "foo.c"; Line 35
#-----
```

The bottom line is that if you want full advantage of function prototypes, you **MUST** use new style function definitions. This seemingly arbitrary limitation was a design decision by the MPW C compiler writers. Their reasoning is now documented in the MPW C 3.1 release notes:

“There already existed a large body of code written in MPW C which used both K&R function declarations and the fact that there was no parameter checking to allow variable numbers of parameters to be passed. If we enforced strict parameter checking, that code would have failed to compile. Denizens of the C world even objected to making these warnings! We therefore do parameter checking only on declarations of the form:

```
int foo (int parm1, int parm2);
```

and not on:

```
int foo (parm1, parm2)
int parm1;
int parm2;
```

Perhaps in the future the -w2 option will at least generate warnings in such cases.”

MPW C 3.1 symbolic table limit

Written: 10/31/90

Last reviewed: 8/1/92

When we compile a file with the MPW C 3.1 “-sym on” option, it seems that the global and local variables (references) cannot exceed 12000. Is that the limit? Are there ways to increase the size?

—

There is no official way to change limit values with symbolic table generation. However, the internal values usually are increased with new tool releases. You might check the latest MPW C 3.2 beta compiler on the current E.T.O. (Essentials-Tools-Objects) CD for limit changes. In general, try to test each module in your code separately. The SADE debugger will work far faster if you have .SYM information generated for the few modules you want to test.

MPW C compiler “-m” option

Written: 2/22/91

Last reviewed: 8/1/92

If I use the “-m” option in the MPW C compiler to build my application, will my application be able to run in the 68000 machines?

—

The -m option doesn't pertain to the processor that you want your application to run on. Rather, it generates less efficient code and allows you to address over 32K of global data. Because you can use over 32K of data, your code has to calculate the address of your data all of the time, hence the linker and your code will run slower. If you type “Help C” in MPW, it should say that the -m option generates 32 bit reference for data.

By default, the C compiler produces code that is compatible with 68000 machines. For additional information, see the Macintosh Technical Note “Stand-Alone Code, *ad nauseam*.”

MPW C 3.2.2 -e option problem

Written: 4/2/92

Last reviewed: 5/21/92

When we use the -e flag with MPW C 3.2.1 and earlier to preprocess a file, if the file contains an #include statement, the included file is inserted in the output in place of the #include statement. For example, if fred.h is

```
#define fred 1
```

and fred.c is

```
#include "fred.h"
main() {}
```

then the command `c -e fred.c` results in

```
#define fred 1
main() {}
```

However, with MPW C 3.2.2 the included file is not inserted and the result is

```
#include "fred.h"
main() {}
```

Is this a bug or is it intentional? Is there any way to reproduce the old result?

—

The -e option malfunctions or doesn't work at all when using MPW C 3.2.2. Until the problem is fixed, we suggest you use MPW C 3.2.1 to do the -e preprocessing. Having done that, you will presumably want to use MPW C 3.2.2 or later to do the actual compilation.

MPW C 3.2 and volatile keyword support

Written: 3/12/92

Last reviewed: 5/21/92

My MPW C 3.2 macros to access memory-mapped I/O on a NuBus card with a 68030 processor, as in the following example:


```
/*
 * Macro:  RdUSCl(reg)
 * Reads 8 bit value from low byte of USC register.
 */
#define RdUSCl(reg) (*(volatile unsigned char *) (reg+TD_PORT+lower_byte))
```

where reg, TD_PORT and lower_byte are all constants. If I now write a function which uses the RdUSCl macro twice in succession, the I/O port only gets read once. In other words, MPW C appears to be ignoring the “volatile” keyword. Is there some way of declaring a variable with a preset address as with other compilers?

MPW C’s volatile keyword support was not implemented in the spirit of ANSI C. It only makes sure the variable is not optimized away (as in the case of placing it in a register or stripping it out). It does not take into account I/O address validity. Volatile keyword support is part of Apple’s future compiler directions. Meanwhile, your best bet is really to make a special get function which reads the register; eventually in-line code would produce a performance-wise quick fetch routine.

Macintosh Quadra performance optimization compiler flag

Written: 6/11/92

Last reviewed: 9/15/92

Are there any C++ or C compilation flags that will optimize performance of the Macintosh Quadra computers? Even when I use the “-NeedsMC68030” flag in MacApp, an investigation of the MABuild source files reveals that it sets compiler flags only for the 68020 optimization. If Quadra-specific compilation flags don’t exist, do you have any Quadra performance optimization suggestions?

The current MPW compilers don’t have a 68040 performance optimization flag, though Apple’s future compilers will optimize code for the best possible 040 performance. In the meantime, here are some tips on 040 performance tuning:

- Cache management for the 040 can give you the biggest performance boost. Keep program loops inside the cache space, and flush the cache as seldom as possible. In most cases you’ll have small loops inside the 4K instruction cache.
- You might get better performance by not calling BlockMove, because the system flushes the cache when you call it in case you’re moving code. If you’re moving data, the cache doesn’t need to be flushed, but the system can’t tell from the BlockMove call whether you’re moving code or data. Testing will help you determine whether you should call BlockMove or write your own transfer routine. The new MOVE16 opcode is used by the BlockMove trap when the system is running on an 040 processor, but because of problems with this opcode in early 040 processors, it requires special handling. For details, see the Macintosh Technical Note “Cache As Cache Can” (formerly #261).
- Transcendental functions aren’t implemented in the 68040 hardware as they are in the 68881 chip used with the 68020 and 68030. Consequently, the functions are emulated in software, resulting in slower performance. If you suspect that your floating point performance is less than optimal, consider modifying your code to use functions supported by the internal 040 FPU. See the Macintosh Technical Note “FPU Operations on Macintosh Quadra Computers”

(formerly #317) for more information about this performance factor. Future MPW compiler and library releases will support faster transcendental operations and floating point-to-integer conversions.