New Technical Notes

Macintosh

®

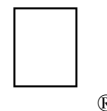## Developer Support

MPW 2.0.2 Bugs
Platforms & Tools

Revised by: Andy Shebanow   October 1988
Written by: Dave Burnard   August 1988

This Technical Note describes latest information about bugs or unexpected "features" in the MPW C, Pascal, and Assembler products and the Toolbox and OS Interface Libraries. We intend this Note to be a complete list of all known bugs in these products, which will be updated as old bugs are fixed, or new ones appear. If you have encountered a bug or unexpected feature which is not described here, be sure to let us know. Specific code examples are useful.

The bugs described in the October 1 revision of this Note will be fixed in the 3.0 release of MPW scheduled for Fall 1988.

Changes since August 1, 1988: Corrected the description of "bug" #3 under MPW C as it is not a bug according to the definition of the C language and corrected an error in bug #2 of the Interface Libraries concerning the glue for `_SlotVInstall` and `_SlotVRemove`.

C Language

The following information applies to the C compiler and its associated libraries shipped with the 2.0.2 version of MPW.

1)   A series of bugs involving floating point array elements and the +=, *=, and = operators. A similar bug was reported as fixed in MPW 2.0.2, unfortunately the fix did not apply to array elements. This bug ONLY occurs when using SANE in combination with `float` or `double` variables, it does not occur if the `-mc68881` compiler option is specified or if `extended` variables are used. The following fragment illustrates the bugs:

```
main()
{
        double x[2],y;  /* Also fails if x,y are declared float
                              but succeeds if declared extended */
        x[0] = 0.5;
        y = 5.0;
        x[0] += 2.0*y;
        printf("x[0] = %f\n", x[0]);
```

```
x[0] = 0.5;
y = 5.0;
```

```
x[0] = 0.5;
y = 5.0;
```

```
                x[0] = x[0] + 2.0*y;
                printf("x[0] = %f\n", x[0]);

                x[0] = 0.5;
                y = 5.0;
                x[0] *= 2.0*y;
                printf("x[0] = %f\n", x[0]);

                x[0] = 0.5;
                y = 5.0;
                x[0] = x[0]*(2.0*y);        /* Succeeds if parenthesis are removed */
                printf("x[0] = %f\n", x[0]);

                exit(0);
        }
```

This code fragment returns the erroneous values 0.5, 0.5, 0.5, 0.5 (the correct values are 10.5,10.5, 5.0 and 5.0).

Workaround: If using SANE, use `extended` variables in these situations.

2)   Taking the address of a floating point formal parameter (function argument) to a function fails. This bug occurs when using either SANE or the `-mc68881` compiler option in combination with `float` or `double` function arguments, it does not occur if the function arguments are declared `extended`. The following fragment illustrates two instances of this bug:

```
#include <Types.h>
#include <Math.h>
#include <stdio.h>

#define real float  /* Fails with either float or double */

main()
{
        Bug1(1.0, 2.0, 3.0);
        Bug2(1.0, 2.0, 3.0);
}

Bug1 (x, y, z)
        real x, y, z;
{
        real *p, *q, *r;

        /* Take address of arguments, assign directly */
        p = &x; q = &y; r = &z;

        fprintf(stderr, "Example 1: Before: %g, %g, %g\n", x, y, z);
        *p = 11.0;
        *q = 12.0;
        *r = 13.0;
        fprintf(stderr, "Example 1: After:  %g, %g, %g\n", x, y, z);
}

Bug2(x, y, z)
        real x, y, z;
{
        fprintf(stderr, "Example 2: Bug2 Before: %g, %g, %g\n",
                x, y, z);
```

```
        /* Take address of arguments, assign indirectly */
        foo(&x, &y, &z);
        fprintf(stderr, "Example 2: Bug2 After:  %g, %g, %g\n",
               x, y, z);
}
foo(x, y, z)
real *x, *y, *z;
{
        fprintf(stderr, "Example 2: foo Before: %g, %g, %g\n",
               *x, *y, *z);
        *x = 11.0;
        *y = 12.0;
        *z = 13.0;
        fprintf(stderr, "Example 2: foo After:  %g, %g, %g\n",
               *x, *y, *z);
}
```

This is, in fact, a general problem with C compilers. The underlying reason for this problem is related to the automatic <u>widening</u> and <u>narrowing</u> of basic types performed by C compilers. For instance in C, as defined by K&R (*The C Programming Language*, Kernighan & Ritchie, 1978, Appendix A Sec. 7.1, p186 and Sec. 10.1, p206), variables of type `char` and `short` are widened to `int` before being passed as function arguments and narrowed before use inside the function. Similarly, the floating point type `float` is automatically widened to `double` before being passed. K&R notes, however, that "C converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declarations <u>adjusted</u> to read `double`." The value of such a formal parameter is **not** narrowed to the declared type of the parameter, instead the declared type is **adjusted** to match that of the widened value. So, in fact, the sample code above will fail if `real` is defined as `float`, even on a bug free K&R conforming compiler.

In MPW C, where `float` and `double` are widened to `extended`, the sample code fails for either `float` or `double` formal parameters. This can, of course, lead to additional problems if you are porting code from an environment where `double` was the widened floating point type (where taking the address of a double formal parameter would work as expected).

Workaround: Taking the address of a function argument is not recommended; you should make a local copy in a temporary variable and take the address of the copy. If you must take the address of a floating point function argument, make sure it is declared as type `extended`, and the pointer is of type `extended*`.

3)   The shift operators `>>` and `<<` can sometimes produce unexpected results when applied to unsigned short and unsigned char operands. The anomaly lies in the fact that the 680x0 `LSR.L` and `LSL.L` instructions are used instead of the `LSR.W` and `LSL.W` or the `LSR.B` and `LSL.B` instructions. The following example illustrates this anomaly:

```
main()
{
        unsigned short u, c;
        short i, k;

        u = 0xFFFF;
        k = 8;
```

```
        i = (u << k)/256;
        printf("unsigned short: i = %d, %#x\n", i, i);

        c = 256;
        i = (u << k)/c;
        printf("unsigned short: i = %d, %#x\n", i, i);
}
```

This code fragment returns the values -1, 0xFFFFFFFF and -1, 0xFFFFFFFF, which are the correct values as defined by the C language, however, you might be expecting 255, 0xFF and 255, 0xFF.

4)   The compiler optimization flags −q (sacrifice code size for speed) and −q2 (memory locations won't change except by explicit stores) can produce incorrect code. We have several vague descriptions of problems, and are looking for more specific examples. The following example illustrates a specific bug that occurs when using <u>enumerated types</u>:

```
badfunc(bool, i)
        Boolean *bool;
        int i;
{
        while (true) {
                if(i < 3) {
                        *bool  = true;
                        if (func(1)) return;
                }
                if (func(i)) return;
        }
}
```

The enumerated type here is the type used to define the values true and false in the header file Types.h. The optimizer is apparently confused by the fact that true has the char value 1, which it thinks is the same as the int value 1 to be passed to the function func(). The object code produced for the two calls to the function func() is:

```
...
MOVEQ  #$01,D3       ; Move the constant 1 into a register
...
MOVE.B D3,(A2)       ; Assign "true" to variable bool
MOVE.B D3,-(A7)      ; Attempt to push integer 1 onto stack
               ; ERROR should be MOVE.L !!!!!
JSR    *+$0002       ; JSR to func
...
MOVE.L D4,-(A7)      ; Correctly push integer variable i onto stack
JSR    *+$0002       ; JSR to func
...
```

In the first function call, the int constant 1 is passed as a byte value! Since the stack is correctly adjusted after the first call this error may go undetected (except that the called function may spot the resulting nonsensical parameter). This problem is, of course, not limited to the enumerated type defining true and false, but can occur as a side effect of any of the many enumerated types defined in the Toolbox header files.

Workaround: The best solution, for now, is to avoid using the optimization flags −q and −q2 altogether.

5) The compiler flag -mc68020 (generate MC68020 instructions) generates inconsistent code when passing structures by value. Specifically, structures larger than 4 bytes that are not a multiple of 4 bytes are padded out to the nearest 4 byte multiple before being pushed onto the stack by the calling routine. Unfortunately, the called routine does not take this padding into account when accessing its function arguments. The following example illustrates this bug:

```
#include <Types.h>
#include <strings.h>

typedef struct        /* 6 byte long structure */
{
  short Temp1;
  short Temp2;
  short Temp3;
} TestStruct;

main()
{
  TestStruct reply;

  foo(reply,"Hello world.\n");
}


foo(tst ,str)
  TestStruct tst;
  char *str;
{
  Debugger(); /* So we can look, before stepping off the cliff */
  printf("%s",str);
}
```

Since function arguments are pushed onto the stack in left to right order in C, the pointer to the string constant "Hello world.\n" is pushed onto the stack before the <u>padded</u> contents of the reply structure. Thus when the called function, foo(), goes to fetch the argument str, it looks at the location just beyond where the argument tst is located. Unfortunately, since the called function does not know the structure argument was padded, it will not find the correct value for the second argument (or in the general case, any arguments following the structure).

Workaround: When using the -mc68020 compiler option, either don't pass structures larger than 4 bytes by value, or make sure all structures larger than 4 bytes are padded out to a multiple of 4 bytes.

6) Switch statements with non-integer controlling expressions can fail. The following example illustrates the problem:

```
#include <stdio.h>
#include <strings.h>
main()
{
        unsigned short w = 1;

        printf("The following test fails in MPW\n");
        printf("Should switch to 1; actually got to ");

        switch (w) {
```

```
        case 1:
                printf("1\n");
                break;
        case 5:
                printf("5\n");
                break;
        case 32771:
                printf("32771\n");
                break;
        case 32773:
                printf("32773\n");
                break;
        default:
                printf("default\n");
                break;
        }
}
```

In this example, instead of reaching `case 1:` in the switch statement, the `default:` case is reached due to the fact that the compiler generates comparison instructions based on word length values. The 1st Edition of K&R (*The C Programming Language*, Kernighan & Ritchie, 1978, Appendix A Sec. 9.7, p202) requires all case constants to have integer type and the controlling expression to be coerced to integer type. The 2nd Edition (ANSI) of K&R (*The C Programming Language*, Kernighan & Ritchie, 1988, Appendix A Sec. 9.4, p223) requires that the controlling expression and all case constants undergo "integral promotion"—promotion to `int` (or to `unsigned int` if necessary). MPW 2.0.2 C fails to promote either the controlling expression or the case constants to type `int`.

Workaround: If the controlling expression is manually coerced to type `int` this example functions correctly.

7) Variable declarations inside the body of switch statements, when combined with the `-g` compiler flag, can cause the compiler's code generator to abort with the message: "Code generator abort code 615." The following example illustrates the problem:

```
foo(i)
        int i;
{
        switch (i) {
                int j;   /* VARIABLE DECLARATION INSIDE BODY OF SWITCH */


                case 0:
                        j = 22 +i;
                        printf("INSIDE: i=%d, j=%d\n", i, j);
                        break;
                case 1:
                        j = 57 -i;
                        printf("INSIDE: i=%d, j=%d\n", i, j);
                        break;
                default:
                        j = i;
                        printf("INSIDE: i=%d, j=%d\n", i, j);
                        break;
        }
}
```

While such a declaration is perfectly legal C, it is a bad practice (*The C Programming Language*, Kernighan & Ritchie, 1978, Appendix A Sec. 9.2, p201). K&R go on to point out that if the declaration of the variable contains an initializer, the initialization will NOT take place. This is also the ANSI draft C standard's interpretation.

Workaround: Since the -g option is a very useful debugging option, move the variable declaration outside the body of the switch statement.

8)   Compatibility Note: Local variable declarations of the form char s[]; (as an array of unspecified length), may not behave as expected. Pre-ANSI compilers often expand such declarations into the equivalent of an extern declaration. MPW 2.0.2 C, and most modern compilers, treat this construct as the declaration of a <u>new </u>array, thus overriding any global declaration with the same name. In the following example

```
/* From file foo.c */
char s[255];

main()
{
        strcpy(s, "This is one heck of a mess");
        otherfunc();
}


/* From file bar.c */
otherfunc()
{
        char s[];

        printf("%s\n", s);
}
```

garbage is printed. As a local variable declaration, this declaration is incomplete since no length is specified or implied, so an ANSI C compiler will of course fail. This is obviously not a recommended programming practice, but if you are porting old C code you may encounter this usage.

Workaround: ALWAYS use the declaration extern char s[]; instead.

9)   Another instance where declarations of the form type s[]; (as an array of unspecified length) may not behave as expected, is as a member of a struct or union. Pre ANSI compilers often expand such declarations into the equivalent of a pointer declaration. This construct is explicitly prohibited in ANSI C structures and unions. MPW 2.0.2 C on the other hand, issues no warning and treats this construct as the declaration of an array of length zero, which occupies no space in the struct. In the following example

```
typedef struct  ST1  {
        int array1[];         /* Zero length array or Ptr to array? */
        int array2[];
        int array3[];
}ST1;

main()
{
```

```
        ST1 s1;
        int i1, i2, i3;

        i1 = s1.array1[0];
        i2 = s1.array2[0];
        i3 = s1.array3[0];
}
```

the three fields of the `struct ST1` are located at the same memory location, and the assignments shown will actually copy garbage into the integers, since <u>no space</u> was allocated for even the first element of the arrays. Unfortunately, structures containing an array of unspecified or zero length as the final member are sometimes used to indicate a structure containing a variable length array. While this may be useful, it is not tolerated by ANSI C and thus is not a recommended programming practice. However, if you are porting old C code you may encounter this usage.

Workaround: ALWAYS use the declarations of the form `type *s;`, `type (*s)[];`, or `type s[1]` (depending on the intended meaning) in structures and unions instead.

10)   The routines `_SetUpA5` and `_RestoreA5` described in the OS Utilities Chapter of *Inside Macintosh*, Volume II, are missing. Refer to M.OV.A5 for two new routines which solve this problem.

11)   Hint: Switch statements with large numbers of cases (over 100 or so) can trigger the appearance of the MPW Bulldozer cursor (signaling heap purging and compacting in progress), can cause "Out of memory" errors from the compiler, or at least take a <u>very long</u> time to compile.

Workaround: Large switch statements can be split up into smaller ones to avoid these symptoms.

Pascal Language

The following information applies to the Pascal compiler and its associated libraries shipped with the 2.0.2 version of MPW.

1)   The Pascal compiler generates incorrect code when accessing the elements of `packed Boolean` arrays with more than 32767 elements. The generated code contains an `ASR.W` instruction to compute the byte offset into the array, this of course fails for numbers larger than 32767 ($7FFF). The following example, which zeroes bits far beyond the end of the array itself, illustrates the error:

```
PROGRAM PackArrayBug;

    VAR
      MyBits:     packed array[0..33000] of Boolean;

      PROCEDURE BadCode;
      VAR
          i: longint;
```

```
            BEGIN
                for i := 0 to 33000 do
                  MyBits[i] := false;
            END;

        BEGIN
                BadCode
        END.
```

Workaround: Don't use `packed Boolean` arrays with more than 32767 elements.

2)   The Pascal compiler fails to detect the situation where a procedure name is passed as an argument to a routine expecting a function as an argument. The following example illustrates the error:

```
PROGRAM FuncArgBug;

        PROCEDURE ExpectsFunc(x: integer; function Visit(y1: longint;
                    y2: char;): Boolean);
        VAR
            result: Boolean;
            yc: char;
        BEGIN
            yc := 'A';
            result := Visit(x, yc);
        END;

        PROCEDURE FormalProc(y1: longint; y2: char;);
        BEGIN
            writeln(y1: 1, ' ', y2);
        END;

        BEGIN
            ExpectsFunc(5, FormalProc);
        END.
```

This type of problem typically leads to stack misalignment and spectacular crashes.

Workaround: Make certain that Pascal routines expecting functions as arguments are indeed passed functions.

3)   The `-mc68881` option causes the Pascal compiler to generate incorrect code when calling external C language routines with floating point `extended` arguments. Apparently the compiler miscalculates the size of the `extended` argument so that it incorrectly removes the arguments from the stack. The following example, which can corrupt the local variables of its caller, illustrates the error:

```
FUNCTION E2S(f: Str255; x: extended): Str255;
   VAR
       dummy: integer;
       i:      integer;
       t:      Str255;

  FUNCTION sprintf(str: Ptr; fmt: Ptr; x: extended):
            integer; C; EXTERNAL;

  BEGIN
       t[0] := chr(0);
       f[ord(f[0]) +1] := chr(0);
```

```
        dummy := sprintf(@t[1], @f[1], x);

        i := 0;
        repeat
          i := i+1;
        until ((t[i] = chr(0)) or (i > 254));
        t[0] := chr(i);

        E2S := t;
   END.
```

The relevant portions of the generated code are:

```
LINK        A6,#$FDF0       ; LINK for local vars
MOVEM.L     D6/D7,-(A7)     ; Save Registers used here
...
...
LEA         $FF00(A6),A0    ; Get address of extended var
MOVE.L      -(A0),-(A7)     ; Push extended onto stack
MOVE.L      -(A0),-(A7)
MOVE.L      -(A0),-(A7)
PEA         $FF01(A6)       ; Push address of format str
PEA         $FDF1(A6)       ; Push address of target str
JSR         *+$0002  ; JSR to sprintf
LEA         $0012(A7),A7    ; Pop args off stack
        ; (SHOULD pop off $14 bytes!)
MOVE.W      D0,D6   ; Save function result
...
...
MOVEM.L     (A7)+,D6/D7     ; Restore registers used here
        ;  (Now they've been corrupted!)
UNLK        A6; Correctly restores A7
MOVEA.L     (A7)+,A0
ADDQ.W      #$8,A7
JMP         (A0)    ; JUMP back to caller
```

Notice that this code would have succeeded if the routine had not used the D6 and D7 registers for storage, and then restored them (incorrectly) before returning.

Workaround: When calling such a routine with the -mc68881 option, isolate the call in a small subroutine or function that has no local variables, so that registers will not need to be saved and restored.

4)   The {$SC+} compiler flag for short circuiting AND and OR operators can sometimes produce incorrect code. The following example does not work if {$SC+} has been used:

```
USES
  Memtypes,QuickDraw,OSIntf,ToolIntf,PackIntf;

VAR
  b1,b2,b3 : BOOLEAN;

Begin
  b1 := false;
  b2 := true;
  b3 := true;

  if not (b1 and b2) and b3 then
```

```
        SysBeep(40);
End.
```

Workaround: Don't use the {$SC+} compiler flag.

5) The Pascal compiler generates incorrect overflow checking code for `longint` valued arguments to the `Odd` function. The generated code contains a signed divide (`DIVS`) by 1 followed by a `TRAPV`, thus the overflow flag is set for values greater than $7FFF. The following example will fail with a `CHK` exception, unless the {$OV+} directive is removed:

```
{$OV+}
PROGRAM PascalOdd;

        VAR
                IsOdd: Boolean;
                longval: LONGINT;

        BEGIN
                longval := 123456;
                IsOdd := Odd(longval);
        END.
```

Workaround: Don't use the {$OV+} compiler flag if you pass `longint` values to the `Odd` function.

6) The Pascal compiler generates incorrect code when functions with `RECORD` type are used as the object of a `WITH` statement. This will only occur if the function is called at a level below the main program, and if the length of the `RECORD` type is 4 bytes or less. The generated code often contains an `LEA  (A7)+,` an instruction, which is of course illegal. The following example demonstrates this unusual situation:

```
PROGRAM PasRecordValFuncBug;

  TYPE
    OurRecord   =
      RECORD
        a:     Integer;
      END;

  FUNCTION RecordValuedFunction: OurRecord;
  BEGIN
  END;

  PROCEDURE ContainsBadCode;

  BEGIN
    WITH RecordValuedFunction DO BEGIN  { This usage bad code. }
      a:=a;
    END;
  END;

BEGIN { PasRecordValFuncBug }
  ContainsBadCode;
  WITH RecordValuedFunction DO BEGIN  { This usage is okay. }
    a:=a;
  END;
END. { PasRecordValFuncBug }
```

Workaround: Don't use RECORD valued functions as the object of WITH statements.

Assembly Language

The following information applies to the Assembler and its associated libraries shipped with the 2.0.2 version of MPW.

1)   There are no known outstanding bugs in the MPW Assembler.

Interface Libraries

The following information applies to the Toolbox and OS interface libraries shipped with the 2.0.2 version of MPW.

1)   The glue for the Device Manager call GetDCtlValue [Not in ROM] described in *Inside Macintosh*, Volume III, is incorrect and will return an incorrect value for the handle to the driver's device control entry. The following is a corrected version of the erroneous glue found in the library file Interface.o:

```
;----------------------------------------------------
;FUNCTION GetDCtlEntry(refNum: Integer) : DCtlHandle;
;----------------------------------------------------
GetDCtlEntry PROC EXPORT
        MOVEA.L      (SP)+,A0              ;Get the return address
        MOVE.W (SP)+,D0           ;Get the refNum
        ADDQ.W #$1,D0        ;Change to a
        NEG.W  D0                    ;  Unit Number
;===>   LSR.W  #$2,D0        ;==Shift in wrong direction!
        LSL.W  #$2,D0        ;Times 4 bytes/entry
        MOVEA.L      UTableBase,A1;Get address of unit table
        MOVE.L (A1,D0.W),(SP)       ;Get the DCtlHandle
        JMP    (A0)          ;And go home
```

This error will affect C, Pascal, and assembly language users.

Workaround: Use the corrected glue for GetDCtlValue.

2)   The glue for the register-based Vertical Retrace Manager calls _SlotVInstall and _SlotVRemove described in *Inside Macintosh*, Volume V, is incorrect. The following are corrected versions of the erroneous glue for these routines found in the library file Interface.o:

```
;-----------------------------------------------------------------------
;FUNCTION SlotVInstall(vblTaskPtr: QElemPtr; theSlot: Integer): OSErr;
;-----------------------------------------------------------------------
SlotVInstall PROC EXPORT
        MOVEA.L         (A7)+,A1      ; save return address
        MOVE.W (A7)+,D0        ; the slot number
        MOVEA.L         (A7)+,A0      ; the VBL task ptr
        _SlotVInstall
        MOVE.W D0,(A7)         ; save result code on stack
        JMP    (A1)   ; return to caller
        ENDPROC


;-----------------------------------------------------------------------
;FUNCTION SlotVRemove(vblTaskPtr: QElemPtr; theSlot: Integer): OSErr;
;-----------------------------------------------------------------------
SlotVRemove PROC EXPORT
        MOVEA.L         (A7)+,A1      ; save return address
        MOVE.W (A7)+,D0        ; the slot number
        MOVEA.L         (A7)+,A0      ; the VBL task ptr
        _SlotVRemove
        MOVE.W D0,(A7)         ; save result code on stack
        JMP    (A1)   ; return to caller
        ENDPROC
```

These errors will affect C, Pascal, and assembly language users.

Workaround: Use the corrected glue for `_SlotVInstall` and `_SlotVRemove`.

3)   The glue for the register based Start Manager calls `_GetTimeout` and `_SetTimeout` described in *Inside Macintosh*, Volume V, is incorrect . The following are corrected versions of the erroneous glue for these routines found in the library file `Interface.o`:

```
;-----------------------------------------------------------------------
;PROCEDURE GetTimeout(VAR count: INTEGER);
;-----------------------------------------------------------------------
GetTimeout PROC EXPORT
;===>   CLR.W    -(A7)              ;===OOPS, selector in A0 not on stack
        SUBA.L    A0,A0            ;Put selector in A0, i.e. 0
        _InternalWait
        MOVEA.L    (A7)+,A1 ;Pop return address into A1
        MOVEA.L    (A7)+,A0 ;Pop location for VAR count
        MOVE.W    D0,(A0)   ;Stuff returned value into count
        JMP       (A1)              ;And go home

;-----------------------------------------------------------------------
; PROCEDURE SetTimeout(count: INTEGER);
;-----------------------------------------------------------------------
SetTimeout PROC EXPORT
        MOVEA.L    (A7)+,A1         ;Pop return address into A1
        MOVE.W     (A7),D0  ;Move count parameter into D0
;===>   MOVE.W     #$0001,(A7) ;===OOPS, selector in A0 not on stack
        MOVEA.W    #$0001,A0        ;Put selector in A0
        _InternalWait
        JMP        (A1)             ;And go home
```

These errors will affect C, Pascal, and assembly language users.

Workaround: Use the corrected glue for `_GetTimeout` and `_SetTimeout`.

Further Reference:
- M.OV.A5