Release Notes

SC and SCpp v. 8.1.02

C and C++ Compilers for 68K Macintosh

Contents

Important: If you had installed the "Symantec For MPW" product (containing SC, SCpp, and the SCBin, SCIncludes, and SCLibraries folders) at some point in the past and prior to installing the MPW environment, you should execute the following command in order to delete the named files/folders:

```
Delete -i -y ∂
"{MPW}"SCBin # (found at the root of the MPW folder) ∂
"{MPW}"Libraries:SCLibraries # (found in the Libraries folder) ∂
"{MPW}"Interfaces:SCIncludes # (found in the Interfaces folder) ∂
"{MPW}"UserStartup•SC # (found at the root of the MPW folder) ∂
"{MPW}"UserStartupTS•SC # (found at the root of the MPW folder)
```

If the two UserStartup files are not removed prior to restarting MPW, the definition for the SCIncludes variable will be reset to an incorrect value. The default definition for SCIncludes is now included in the main MPW "Startup" script in the MPW environment. Therefore, the two UserStartup≈•SCscripts are obsolete.

Introduction

Beginning with ETO/MPW Pro #16, the Symantec SC and SCpp compilers replaced the MPW C and CPlus compilers previously used to compile code for the 68K Macintosh. The SC and SCpp compilers support development for both the Classic 68K runtime and the CFM-68K runtime architectures. They are considerably faster than the compilers they replace, and generate code of comparable quality. All of the Classic 68K runtime libraries included in MPW have been updated to support the calling conventions of the SC and SCpp compilers.

In addition to SC and SCpp, the Symantec preprocessor, SCpre, has also been included. It can be used to check source code for preprocessor errors without running the full compiler.

New Features

- SC and SCpp are now fat applications.
- There are no limitations on developing 68K objects on a Power Macintosh except for the range of long doubles. Long double exponents are limited by the ddrt library to 0 < e < 2047 instead of 0 < e < 32766. This limitation will be corrected in the future. Also, the bit patterns for NAN, NANS, and -0.0 might differ between objects generated on a Power Macintosh and a 68K Macintosh.
- The compilers now support a new option setting that allows control over included files. It currently can only be controlled using a #pragma as follows:

  ```
  #pragma options(system_includes_from_project_tree)
  ```

  This switch defaults to off and it controls the behavior of includes appearing within <>. If this option is turned on, #include statements using the <> behave identically to #include statements using “”. For example:

  ```
  #include <stdlib.h>      //This #include works as for <> includes
  #pragma options(system_includes_from_project_tree)
  #include <stdio.h>       //This #include works as for “” includes.
  ```

- Support for not placing variables into registers when calling setjmp() and other functions which save and restore register contents. There is a new #pragma which is used to indicate functions with this property. It is the same name and semantic as the THINK C #pragma nooptimize (XXX):

```
#pragma nooptimize (setjmp)     //Any function that calls setjmp
                                // will not have variables packed
                                // into registers
```

Note: This allows many implementations of exception handling in C and C++ to be unconcerned with the packing of variables into registers.

- The MPW compilers will resolve aliases to header files placed within directories being searched.

SANE comparison operators

The following mapping of extended floating point comparison operators for SANE is not in the current documentation:

Operator	SANE compare
!>	<=
!<=	>
!>=	<
!<>	==
<>	!=
<>=	none (assumed always TRUE)

Pre-defined Variables

When using `-elems881` (which must be used in conjunction with `-mc68881`), the macro `ELEMS881` is pre-defined so that its presence can be checked for during conditional compilation.

CFM-68K Limitations

Under either `-model cfmseg` or `-model cfmflat`, all floating point types are passed to variable argument functions (e.g., `printf`) as type `double`. Therefore, there may be a loss of precision when programs that used the `extended` type pass such variables to variable argument functions. Note that floating point constants, e.g., 1.0, may have differing interpretations depending on the intended platform for the source code. If it is Power Macintosh code, the interpretation is `double`; if it is classic 68K code, the interpretation is `extended`. A programmer can force the use of `extended` in `printf` by using a cast and the correct format specification, e.g.,

```
printf ("%Lf", (extended) (1.0 + foo) );
```

Incompatibilities between SC/SCpp and MPW C/CPlus

- SC and SCpp use a different calling convention than MPW C and CPlus. In particular, parameters of type char and short are passed differently. In addition, variables and parameters of type int may be either two or four bytes in length when using SC. If two- byte ints are selected as a compiler option, you will have problems passing parameters of type int as well. When using the CFM-68K runtime architecture, parameters of float, double, and long double (extended) are also passed as different sizes than with MPW C. As a consequence, routines compiled by the two compiler families cannot call each other. Although such cross family calls will work if these specific variable types are avoided, mixing the two code families is not recommended.

- Assembly language functions which call or are called by MPW C will need to be updated to use SC/SCpp calling conventions. The calling conventions for Classic 68K runtime and CFM-68K runtime differ. The calling conventions are described in the document Mac OS Runtime Architectures.

- The SC/SCpp-compatible libraries provided with MPW expect arguments of type int to be four-byte values. They will not work properly if you compile your code with two-byte ints.

- SCpp and CFront use different name mangling algorithms and object models. This may cause unresolved symbols when attempting to link together objects compiled with the two compilers. Even if objects can be linked, they will most likely not run because of differences in the object dispatch methods. Do not link together objects compiled by SCpp and MPW CFront.

- Do not mix old style K&R function definitions with new style ANSI prototypes when compiling with `-model cfmseg`. Doing so will generate errors of the type `"#Error: type of 'x' does not match function prototype"`, because of changes in the way implicit conversion of certain types is handled. The following example generates this error;

```
extended foo( extended x );   /*  new-style prototype */

extended foo(x)         /* old-style function definition */
extended x;
{ … }
```

Porting Code from MPW C++ to SCpp

Errors

- **cannot implicitly convert**—This error arises when a class `C` has a conversion operator and a `const` conversion operator to the same type, i.e. `operator X();` and `operator X() const;`. The MPW C++ compiler, in processing an implicit conversion of an instance of class C to an instance of X, is able to resolve this ambiguity based on the "constness" of the instance of C. SCpp, on the other hand, calls this an error, and requires that the user write an explicit cast: `(X) Cinstance` or `X(Cinstance)`.

  There is no certainty as to whether this is really a bug in SCpp, or permissiveness on the part of CFront. The Annotated C++ Reference Manual is not clear on this issue.

- **illegal cast**—This error can also be avoided by writing an explicit cast. It is believed to arise in similar circumstances as the previous one.

- **no return value for function foo()**—This error is emitted when not every alternate exit from a function's code has an assigned value. An instance which might be considered to be a bug is the following:

```
enum q (one, two);
short foo(q x)
{
      switch () {
      case one: return 0;
      case two: return 1;
      }
}
```

The SCpp does not realize that all cases have been covered. The workaround is to add a default case and give it a return value.

Warnings

- **Possible unintended assignment**—instances of code like

```
if (x = whatever() ) {...}
```

  will produce this warning. The compiler assumes that "==" was intended.

- **Casting from incomplete structure type**—This warning does not appear to be significant in most cases. It typically occurs when casting a base class to a derived class.

Requirements on Calls from Pascal

When MPW Object Pascal encounters the construct "External; C;" it generates a call to the named function using MPW C calling conventions. Because these calling conventions are not the same as those used by the SC compiler, this technique will not work when accessing functions compiled with SC. Apple has no plans to update MPW Object Pascal to support the SC calling conventions. In order to access a C function from Pascal, the function should be defined as using Pascal calling conventions , e.g.:

```
pascal int foo (short bar);.
```

Known Outstanding Bugs in SC

Note: Some of the bugs listed in the section after this one, Known Outstanding Bugs in SCpp, also apply to SC.

- When compiling without optimizations, a construct of the form `*<string literal>` or `<string literal>[0]` incorrectly references a word instead of a byte.

  Example: the expression `*"\p3135551212"+1`.

  Workaround: Copy the value into a local char, and then use, or compile with `-opt`

`all`.

- In some cases, the lib or link tool will report

```
### lib: Bad promotion of an object to a type object.
# file=':foo.c.o' id=16401
```

  This is caused by duplicate symbolic information produced by the compiler. The workaround is to compile the file with `-sym off`.

- In some cases, the SC compiler will allocate an odd-size struct as an odd size. This can result in a MOVE.L instruction on an odd offset, causing a crash on 68000. The compiler should be padding the structure to an even boundary.

  The Macintosh Toolbox headers do not contain any such structures.

  Workaround: Manually pad the end of any odd-size structure containing only chars. Manually pad any nested structures containing only chars to start at an even offset within the enclosing structure.

- SC doesn't support all of the pragmas that are supported by MPW C. The missing pragmas include (but aren't limited to)

```
#pragma force_active
#pragma push
#pragma pop
```

- The response to the operators <>= and `!`<>= may not meet NCEG specifications with respect to signalling "invalid for unordered." These operators also may be optimized out to have a value of 1; see the workaround to follow:

  In the following function that uses the NCEG <>= operator, code for all statements following the statement that has the <>= are incorrectly optimized out as if the statement had read if (1) instead of if (x <>= y)

```
void hohum ( void );
void dodah ( void );
extended x, y;
int main ( void )
{

  if ( x <>= y ) return 3;
  dodah ();
  return 2;
}
```

  Workaround:

  Use a construct such as `if ( x )` to suppress overoptimization.

  Example:

```
if ( x )
if ( x <>= y ) return 1;
```

The same problem and same workaround apply to the NCEG !<>= operator.

- SC does not generate a MacsBug symbol when a function is optimized to not require a stack frame.

  Workaround: Specify `-frames` to force a stack frame.

Known Outstanding Bugs in SCpp

- Unlike CFront, SCpp does not place all static constructors in one special segment. In the worst case, program initialization may require that all segments are loaded before the application reaches its main procedure.

- The SC, SCpp, MrC, and MrCpp compilers do not properly implement `#error`. According to ANSI, the directive should cause the compiler to produce a `#error` directive in the following form:

  ```
  #error "This won't work"
  ```

  Instead, these compilers simply produce a message in the following form:

  ```
  # File foo.c; line 17 # Error
  ```

- SCpp bases import-export status on forward references over class definitions. If a class definition is bracketed in `#pragma lib_export on/off` statements, but a forward reference to that class is encountered before the class definition itself, the class will not be marked as exported unless the forward reference is bracketed with the pragmas.

- Cannot implicitly convert from `char*` to signed `char*`

  Workaround: Use typecheck relaxed. The compiler is enforcing ansi rules in this area. The better solution is to add an explicit type cast where needed. The strictly proper solution is to pass a variable of the proper type, but for string constants, this is not always practical.

- SC(pp) generates Global Data for static initializers. Example:

  ```
  pascal short LibFunction (long a)
  {     auto Rect   bounds = {15, 15, 85, 185};   ...;}
  ```

  When compiled with SC will generate a reference to a global `_TMP0`, containing the top/left/right/bottom values.

Workaround: Initialize the struct members individually.

```
bounds.top = 15; ...; bound.right = 185;
```

- The Macsbug labels generated for nested classes are not consistent with the unmangler provided with MacsBug and ResEdit's disassembler.

- When using SCpp in Classic mode to build standalone code resources, if the main routine is declared to use pascal calling conventions and is named `main`, the compiler silently compiles the routine using C calling conventions instead. The problem is detected at link time, when you specify the main entry point and receive an error saying that "main" doesn't exist.

  Workaround: Use a name other than "main."

- SCpp requires that a class be declared before it can be used in a `friend` declaration.

- SCpp generates bad code for `char(x)` type of casting in a switch. The compiler is treating the cast as a declaration. A local `x` is being created and is being loaded, rather than the global `x`. The local `x` is an unitialized value. For example,

  ```
  switch (char(x)) {...}
  ```

  Workaround: Use:

  ```
  c = char(x);
  switch (c) {... }
  ```

Manual Errata

- The following pragma was omitted from the SC/SCpp manual:

  ```
  #pragma template_access segmentname
  ```

  DESCRIPTION

  The segment pragma controls the name of the segment for the generated code. Object code for functions following the pragma segment directive will be assigned to the specified segment. A pragma segment directive should not be placed within a function definition.

  DEFAULT

  The default segment name is `Main`.