White Paper

ZiLOG Z8 Encore!® Compiler Compliance With ANSI STANDARD C

WP000801-0904

ZiLOG Worldwide Headquarters • 532 Race Street • San Jose, CA 95126
Telephone: 408.558.8500 • Fax: 408.558.8300 • www.zilog.com

Abstract

The purpose of this document is to explain differences between the ZiLOG XTools C compiler for the Z8 Encore!® processor family and the ANSI C Standard. These differences consist of both extensions to the ANSI standard, and deviations from the behavior described by the standard.

The particular version of the XTools Z8 Encore! compiler described in this document is the version released in ZDS II release 4.9.0. For the most part, the issues discussed in this document are not expected to change greatly from one compiler release to the next. The final section of the document describes any recent changes.

The ANSI Standard

The ZiLOG XTools Z8 Encore! compiler is a freestanding ANSI C compiler, complying (except as described below) with the 1989 ISO standard which is also known as ANSI Standard X3.159-1989.

Freestanding Implementation

A "freestanding" implementation of the C language is a concept defined in the ANSI standard itself, to accommodate the needs of embedded applications which cannot be expected to provide all the services of the typical desktop execution environment (which is called a hosted environment in the terms of the standard). In particular, there is presumed to be no file system and no operating system.

The use of the standard term "freestanding implementation" means that the compiler must contain, at least, a specific subset of the full ANSI C features. This subset consists of those basic language features appropriate to embedded applications. Specifically, complex numbers are not required for a freestanding implementation, and the list of required header files and associated library functions is minimal, namely: `<float.h>`, `<limits.h>`, `<stdarg.h>`, and `<stddef.h>`. A freestanding implementation is allowed to additionally support all or parts of other standard headers, but is not required to. The XTools compiler, for example, supports a number of additional headers from the standard library, as specified in section “Library Files Not Required for Freestanding Implementation” on page 13.

One minor deviation from the standard which is due to the XTools compiler being a freestanding implementation is the prototype for main(). This is discussed below in section “Prototype of Main” on page 10.

Extensions Allowed

A "conforming implementation" (i.e. compiler) is allowed to provide extensions, as long as they do not alter the behavior of any program that uses only the standard features of the language. The ZiLOG XTools Z8 Encore! compiler uses this concept to provide language extensions that are useful for developing embedded applications and for making efficient use of the resources of the Encore! processor. These extensions are described in section “The 1999 Standard” below.

The 1999 Standard

The 1989 standard with which the XTools compiler is meant to be compliant is the older and better known of the two existing ANSI/ISO C standards. The later 1999 standard, also known as ISO/IEC 9899:1999, adds many features to the C language, but most of these are little known; in fact, most of these features are not yet supported in one of the most popular desktop C/C++ compilers. Most of these features are also not supported by the XTools Z8 Encore! compiler.

Supported New Features from the 1999 Standard

C++-Style Comments. Comments preceded by a // and terminated by the end of a line, as in C++, are supported.

Trailing Comma in Enum. A trailing comma in enum declarations is allowed. This essentially codifies a common syntactic error that does no harm. Thus, a declaration such as

```
enum color {red, green, blue,} col;
```

is allowed (note the extra comma after "blue").

Empty Macro Arguments. Preprocessor macros that take arguments are allowed to be invoked with one or more arguments empty, as in this example:

```
#define cat3(a,b,c)  a b c
printf("%s\n", cat3("Hello ", ,:World"));
//                           ^ Empty arg
```

Long Long int Type. The type long long int is allowed. (In the XTools Z8 Encore! compiler, this type is treated as the same as long, which is allowed by the standard.)

Removed Implicit Function Declaration. That is, calling a function with no prototype is no longer allowed in the new standard.

By default, the XTools Z8 Encore! compiler enforces this standard. This enforcement can be turned off by deselecting the "Strict ANSI conformance" option.

Except for coping with massive poorly written legacy code, it should never be necessary to deselect this check box. Requiring function prototypes is highly desir-

able as it detects errors at compile time, which is much quicker than detecting them by debugging.

New Features from the 1999 Standard Not Supported

As mentioned above, there are many new features of the 1999 standard that are not supported by the XTools Z8 Encore! compiler, most of which are little known and little used. We mention just a few of these features explicitly here because they are widely used.

Inline Functions. The inline keyword for function declarations is well known since it is standard in C++, and is now also added to the newer 1999 C Standard. We do not support this addition to the language in the XTools Z8 Encore! compiler. Inline functions give greater execution speed at the cost of a significant increase in code size. This cost greatly outweighs the benefit for a processor like the Z8 Encore! with its limited code memory.

Mixed Declarations and Code. The new standard allows variables to be declared anywhere inside a function, rather than just at the start of a block; again, this is familiar coding practice because it has always been standard in C++. Likewise, the C++ style statement

```
for (int i = 0; i < 10; i++) ...
```

is permitted by the new standard. Currently the XTools Z8 Encore! compiler does not support either of these types of declarations.

_Bool type. The new _Bool type is not supported in the XTools Z8 Encore! compiler. However, we note that the desired effect can be obtained simply by adding a line such as

```
typedef unsigned char _Bool;
```

ZiLOG Extensions to the Standard

Address Space Specifiers

The keywords near, far, and rom are used to specify memory spaces in which data may reside. The near and far keywords are therefore useful for controlling which data goes into the "near" area which can typically be accessed with shorter machine instructions, contributing to the efficiency which is a prime concern of many embedded applications. Similarly, the rom keyword allows the user to partition his data storage between RAM and ROM in the way that best fits the needs of his application. In short, these keywords act like the storage class specifiers provided in the Standard. All the other features of their behavior follow from this fact.

Near Keyword

`near` - specifies that data is to be placed at a RAM address less than or equal to 0xFF.

Far Keyword

`far` – specifies that data is to be placed at a RAM address in the range 0x100 to 0xFFF.

Note: Note: It is not recommended practice to mix near and far pointers in operations. It is preferable to consider the near and far address spaces as separate and independent. The fundamental reason for this is that if one declares a near and far pointer and assigns them the same value:

```
near char* np = 0xff;
far  char* fp = 0xff;
```

one still cannot be certain that np and fp point to the same address. This is true because in the Z8 Encore! architecture, the full, 12-bit address of the near pointer depends on which working register page is in use -- in other words, on the value of the RP (register pointer) register. The RP is typically initialized in a way that would make np and fp in this example point to the same address, but this cannot be guaranteed by the compiler.

Rom Keyword

`rom` – specifies that data is to be placed in ROM, or program space, addresses less than or equal to 0xFFFF.

Usage

These keywords are used exactly as the ANSI keywords const and volatile; for example:

```
   rom  char* p; // A pointer in ordinary memory to a character in
rom
   char rom*  p; // Same as above
   char * rom p; // A pointer in ROM memory to a character in
ordinary memory
   char rom* rom p; // A pointer in ROM memory to a character in
ROM memory
```

The `near` and `far` keywords are used in the same way as the `rom` keyword in these examples.

String Placement Syntax

As described in section “Address Space Specifiers” on page 4, the extension keywords near, far, and rom provide the useful service of giving the embedded system developer control over what type of memory space is used to store program variables. However, in many embedded applications there is another type of program data which is often a primary consumer of data space, namely string constants (literals) such as "mystring". The storage location of these constants cannot be controlled using the near, far, and rom keywords because they are not variables.

To allow the user the same control over where to allocate storage for such strings, a special syntax for declaring them is provided. If the following language extensions are not used, they are stored in initialized memory as specified by the memory model (in near memory for the small model, in far memory for the large model).

Near String Constants

N”mystring” : Near string constant. Stores the string in RDATA (near memory). The address of the string is a near pointer.

Far String Constants

F”mystring” : Far string constant. Stores the string in EDATA (far memory). The address of the string is a far pointer.

Rom string constants

R”mystring” : ROM string constant. Stores the string in ROM. The address of the string is a rom pointer.

Usage

The following code snippet illustrates the use of this syntax.

```
#include <sio.h>

void funcn(near char *str)
{
    while (*str)
            putch(*str++);
    putch('\n');
}

void funcf(far char *str)
{
    while (*str)
            putch(*str++);
```

```
    putch('\n');
}

void funcr(rom char *str)
{
    while (*str)
            putch(*str++);
    putch('\n');
}

void main(void)
{
    funcn(N"nstr");
    funcf(F"fstr");
    funcr(R"rstr");
}
```

Static Frames

The Z8 Encore! compiler supports both static and dynamic call frames.

In the more familiar dynamic frames, local variables are stored in memory which is allocated dynamically on the processor stack. Dynamic frames are fully ANSI compliant, and are required to support certain C features -- specifically: recursive functions, functions with a variable number of parameters, and functions called through a pointer.

In static frames, local data are stored in a statically allocated location reserved for each particular function. The linker, however, attempts to reduce the total storage requirements for these frames by overlaying the areas for separate functions that cannot be simultaneously active. This generally results in significantly more efficient code on the Z8 Encore! processor than dynamic frames, for several reasons. Primarily, since the compiler knows the absolute addresses of local variables it can generate more compact code for accessing them than if they must be accessed by offsetting from a frame pointer (as required when using dynamic frames). However, the user must either avoid using the features listed above that require dynamic frames, or must use the reentrant keyword to force the compiler to use dynamic frames for specific functions (see section 2.4).

Unlike other features described here, static frames are selected by setting a compiler option (typically through an IDE control) when compiling the program, rather than by a keyword in the language itself.

The Reentrant Keyword

When using static frames, functions that require dynamic frames should be declared using the reentrant keyword:

```
reentrant void foo(int a, ...);
```

Such functions will be compiled and called using dynamic frames. Functions that require dynamic frames include:

- Any recursive function, including indirect recursion.
- Any function called through a pointer.
- Any function with a variable number of parameters.
- Any function that might be called by an interrupt handler, unless it takes no parameters and has no local non-static data.

When using dynamic frames, the reentrant keyword may be used, but it has no effect.

The Interrupt Keyword

Functions to be used as interrupt handlers are declared with the interrupt keyword, as follows:

```
void interrupt handler (void);
```

The compiler will ensure that functions declared with this keyword meet several criteria for safety of interrupt handling. Specifically: they will save and restore necessary processor state upon entry and exit; they will do so in an atomic manner which is guaranteed to proceed safely in the event of further interrupts occurring; and they will use an alternate return mechanism which returns control to the interrupted function and re-enables interrupts.

Since they are designed to be entered through the process of the interrupt occurring rather than being explicitly called, interrupt handlers must take no arguments and return no value, as shown above.

Note that in the Z8 Encore! interrupt controller, interrupts are disabled in the processor hardware when the interrupt is recognized. Further interrupts therefore cannot be recognized during the execution of an interrupt handler, unless the programmer explicitly does so by including the EI instruction in the handler.

If static frames are in use, functions declared as interrupt handlers are not by default made reentrant.

Char-Sized Enums

By default, enumeration types have the size of ints. Often the enumeration will fit into a character, and treating it as a character will generate more efficient code. To do this, use the syntax

```
enum {RED,GREEN,BLUE} char pcolor;
```

Embedded Assembly

The ZiLOG Z8 Encore! compiler supports embedding of assembly code inside C. This technique should be used only with great caution as the compiler cannot detect and warn of any potentially dangerous constructs in the assembly code, such as register assignments or usages that would interfere with those made by the compiler. As the XTools Z8 Encore! compiler has become more aggressively optimizing over the last several releases, use of embedded assembly has become both more hazardous and less necessary.

There are two methods of inserting assembly language within C code. The first uses the #pragma feature of ANSI C with the following syntax:

```
#pragma asm "<assembly line>"
```

This #pragma can be inserted anywhere within the C source file. The contents of <assembly line> must be legal assembly language syntax, and the programmer is responsible for verifying this correctness. The usual C escape sequences (such as \n, \t, and \r) are properly translated. Otherwise the compiler does not process the <assembly line>; except for these escape sequences, it is passed through the compiler verbatim.

The second method of inserting assembly language is using the asm statement. An asm statement is allowed only within the body of a function and takes the following form:

```
asm("<assembly line>");
```

The asm statement cannot be within an expression.

The Strict ANSI Compiler Option

We mention this Xtools Z8 Encore! compiler option here because its name suggests that selecting it would enforce complete adherence to the ANSI standard and therefore disable all the features we have discussed in this section. However, it does not do so and perhaps should be renamed.

In reality, the action of this option is to enforce adherence to a variety of provisions of the standard, related to function prototypes, pointer comparisons, and other areas. When this option is selected, the following restrictions are then enforced:

The compiler requires prototypes or previous declarations of all called functions, as mentioned in “The 1999 Standard” on page 3. The compiler issues a warning if old-style function declarations ("K & R-style" rather than ANSI) are used. The compiler issues a warning if the code redefines a previously defined macro, including the standard macros defined in the standard header files. Pointers to different types may not be compared. Pointers that do not match as to storage class and qualification (const, volatile, rom, etc.) may not be compared, subtracted, or substituted as function parameters for non-matching pointer types. Pointers may not be compared to integers without an explicit cast. Signed and unsigned integers of the same size are not treated as equivalent in type comparisons.

Deviations from the Standard

There are a small number of areas in which the XTools Z8 Encore! compiler does not behave as specified by the Standard. For the most part, these are areas in which the behavior called for by the standard is not appropriate for embedded applications, but for which exceptions are not allowed in the definition of a free-standing implementation. We describe those areas in this section.

Prototype of Main

For compatibility with hosted applications, the XTools Z8 Encore! compiler uses main() as the function called at program startup. Since the Z8 Encore! provides a freestanding execution environment, there are a few differences in the syntax for main(). The most important of these is that in a typical small embedded application, main() never executes a return as there is no operating system for a value to be returned to, and is also not intended to terminate. If main() does terminate and the standard ZiLOG Z8 Encore! C startup module is in use, control simply goes to the statement:

```
_exit:
    JR _exit
```

For this reason, in the XTools Z8 Encore! compiler main() should be of type void; any returned value is ignored. Also, main() is not passed any arguments, specifically including the standard parameters argc and argv which are allowed in one of the two forms of main() defined by the standard.

In short, the prototype for main() is

```
void main (void);
```

unlike the standard in which the closest allowed form for main is

```
int main (void);
```

ANSI Promotions Disabled

The ANSI standard requires that integer variables smaller than ints always be promoted to ints prior to any computation. So a function such as:

```
char makeUpper(char c)
{
    if (c >= 'a' && c <= 'z')
    c = c - ('a'-'A');
    return c;
}
```

is to be compiled as though it had been written:

```
char makeUpper(char c)
      {
          if ((int)c >= (int)'a' && (int)c <= (int)'z')
              c = (char)((int)c - (int)('a'-'A'));
          return c;
      }
```

On an eight bit processor such as the ZiLOG Z8 Encore!, such promotions are quite inefficient and rarely change the result of a computation. These promotions are disabled by default, but may be enabled by the ANSI promotion compiler option.

The Const Keyword and ROM

The XTools Z8 Encore! compiler provides an option to place const values in ROM memory, which is often used by programmers who desire this level of control over the limited data storage available in the Z8 Encore! memory space. When this option is selected, the treatment of the const keyword is emphatically non-ANSI. Specifically, when this option is selected, the keyword "const" is treated as equivalent to "rom". Then the function prototype

```
foo (const char* src);
```

would imply that the src string is in ROM memory, while under the ANSI standard the src string is in ordinary memory but the function foo() is not allowed to modify *src.

The problem arises when other code calls a function like foo() with an argument string that was not declared const. Under the standard, this is common practice and the programmer's expectation is simply that foo() would refrain from modifying its argument. However, the Z8 Encore! uses different machine instructions for accessing its different memory spaces. The instruction for accessing data in ROM is LDC, which will be generated in the assembly code for foo() in this example. To access data in ram, however, the LD or LDX instructions must be used. Therefore, when a string that was not declared const is passed to foo(), the code for foo() is

unable to access its parameter. Fortunately, the compiler will report an "Incompatible data types" error for code that makes calls of this type.

The effect of this deviation from the standard is primarily that in code which must be portable for all options of the compiler and linker, such as the source code for library functions provided by XTools, parameters may not be declared const.

On new applications, we discourage this use of the const keyword to place data in ROM. Rather, we recommend declaring constant data such as tables using the rom keyword instead. Where portability is a consideration, this can easily be done by preprocessor macros. For example:

```
#ifdef __EZ8__
#  define ROM rom
#else
#  define ROM const
#endif

ROM struct TableElement[] table = { /* stuff */};
```

Const Correctness in the Standard Header Files

In general, our header files are not const correct due to the issue raised in section “The Const Keyword and ROM” on page 11. For example, in our library strcpy is (effectively) declared

```
char* strcpy( char* dst, char* src);
```

while the ANSI standard requires

```
char* strcpy( char* dst, const char* src);
```

As noted above, use of the const keyword here would cause compile-time errors if the CONST=ROM compilation option were selected and then strcpy() was called with an argument for src which had not been declared const.

Double Treated as Float

The XTools Z8 Encore! compiler does not support a double-precision floating-point type. The type "double" is accepted, but is treated as if it were "float".

Areas Not Defined by the Standard.

Certain areas of the behavior of a compiler are left by the standard to the discretion of the compiler implementer. This section describes the specific behavior of the ZilOG XTools Z8 Encore! compiler in some of those areas.

Sizes of Basic Types

The ANSI standard does not specify the sizes of the basic types, except that each type should be at least as large as the next "smaller" type.

In the XTools Z8 Encore! compiler, an int is (by default) the same size as a short, 16 bits, and a char is an 8 bit value. Both long and long long integers are implemented as 32 bit values.

This implementation treats chars as signed values. The standard allows chars to be treated as either signed or unsigned when simply declared as "char"; the compiler implementation is just required to pick one or the other and adhere to that choice. Also, in the XTools Z8 Encore! compiler a wide character, wchar_t, is the same as a short.

Floats, doubles, and long doubles are all implemented as 32-bit IEEE floating-point numbers, what would be a float in most hosted compilers. As noted above in section “Double Treated as Float” on page 12, this is not in strict conformance with the ANSI standard.

There are compiler options to use non-default sizes for some of the int types. If any of those options are selected, the user must rebuild all the runtime libraries, as the libraries supplied with the compiler are built with the default integer sizes. Use of these options is discouraged in applications that call the standard libraries, as it is difficult or impossible to test correct operation of the libraries with all the possible combinations of integer sizes.

Library Files Not Required for Freestanding Implementation

As noted in section “Freestanding Implementation” on page 2, only four of the standard library header files are required by the standard to be supported in a freestanding compiler such as the XTools Z8 Encore! compiler. However, the XTools Z8 Encore! compiler does support many of the other standard library headers as well. The supported headers are listed here. The support offered in our libraries is fully compliant with the Standard except as noted here, and except for the issue of const correctness as described in section “Const Correctness in the Standard Header Files” on page 12.

<assert.h>

<ctype.h>

<errno.h>

<math.h>

Our implementation of this library is not fully ANSI compliant in the general limitations of our handling of floating-point numbers: namely, we do not fully support

floating-point NAN's, INFINITY's, and related special values. These special values are part of the full ANSI/IEEE 754-1985 floating-point standard which is referenced in the ANSI C Standard.

<stddef.h>

<stdio.h>

We support only the portions of stdio.h that make sense in the embedded environment. Specifically, we define the ANSI required functions that do not depend on a file system. For example, printf and sprintf are supplied but not fprintf.

<stdlib.h>

This header is ANSI compliant in our library except that the following functions of limited or no use in an embedded environment are not supplied:

```
strtoul()
_Exit()
atexit()
```

Recent Changes

In this final section, we note for the convenience of our users significant changes that have been made in the ANSI conformance of the ZiLOG XTools Z8 Encore! compiler in recent releases.

String Placement Syntax

The string placement syntax described in “String Placement Syntax” on page 6 is a new feature with the release of ZDS II version 4.9.0 for the Z8 Encore!.

Struct Returns

Prior to the release of ZDS II version 4.9.0, the XTools Z8 Encore! compiler did not support functions that return a structure rather than a basic type or pointer. As one consequence of this restriction, the div and ldiv standard library functions were not supplied. This deviation from the standard has been removed with release 4.9.0.

Note that returning a struct, especially a large one, remains an expensive operation and is often regarded as a practice to be avoided in designing embedded applications.

Char-Sized Enums

The ability to declare enumerations to have the size of char rather than int, described in “Char-Sized Enums” on page 9, is not new in release 4.9.0 but is newly documented.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, contact:

ZiLOG Worldwide Customer Support Center
532 Race Street
San Jose, CA 95126
USA
Telephone: 408.558.8500
Fax: 408.558.8300
www.zilog.com

Information Integrity

The information contained within this document has been verified according to the general principles of electrical and mechanical engineering. Any applicable source code illustrated in the document was either written by an authorized ZiLOG employee or licensed consultant. Permission to use these codes in any form, besides the intended application, must be approved through a license agreement between both parties. ZiLOG will not be responsible for any code(s) used beyond the intended application. Contact the local ZiLOG Sales Office to obtain necessary license agreements.

Document Disclaimer