White Paper

Using the ZiLOG XTools eZ80Acclaim!™ C-Compiler

WP000501-0104

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

Abstract

The Xtools eZ80Acclaim!™ C compiler is an optimizing compiler for C code based on the ANSI C standard, with modifications to support specialized needs of the embedded developer. Basic information about the use of the compiler and details of how it supports particular eZ80Acclaim!™ features are given in the ZiLOG Developer Studio II User Manual. This white paper is oriented toward helping users make the most effective use of the compiler by describing how to deal with some specific issues that often arise in customer usage. We place a special emphasis on practical tips about using the compiler to create efficient code with a small footprint.

We will begin with a short description of how the Xtools compiler differs from a pure implementation of the ANSI C Standard; this section should clarify exactly what the compiler is. Then we will talk about how to use the compiler, giving guidelines for writing code and creating projects that make good use of the compiler.

Embedded Modifications to the ANSI C Standard

In most ways the Xtools eZ80Acclaim!™ compiler is simply an implementation of the ANSI C Standard. However, tailoring a development tool to the needs of the embedded system developer means making a few changes to the standard. Some of these are extensions to the standard which add special capabilities for the embedded arena, and others are restrictions – areas where the standard calls for features that are unnecessary or too bloated to use in embedded applications.

The eZ80Acclaim!™ compiler provides the language extension keyword interrupt to make interrupt handler development easier. This keyword is available only as a function qualifier, for example:

```
void interrupt my_handler (void)
```

The compiler responds to the interrupt keyword by automatically generating code to save machine state on function entry and restore it on function entry. Since interrupt handlers are not explicitly called as other functions are, they cannot take arguments or return values; both the parameter list and return type must be void. See the *ZDS II User Manual -- eZ80Acclaim!*™ (UM0144) for more details.

Another extension to the C standard is the ability to embed eZ80Acclaim!™ assembly code inside a C program. This makes it easy to create a project in which only certain performance-critical procedures, or even critical sections of a function, are coded in assembly while the bulk of the application is developed in C. The *ZDS II User Manual -- eZ80Acclaim!*™ (UM0144) describes two methods for embedding assembly code in a C file.

There are several areas in which the Xtools eZ80Acclaim!™ compiler by design does not support the full ANSI Standard. The largest group of these is the omission of parts of the Standard Library which are not useful for embedded applications, such as those relating to file I/O and other services that would typically be provided in a desktop operating system. This type of limitation is common enough that ANSI actually designates a class of compilers with the term free-standing implementation, to indicate that they are intended to be used in an environment where the services of a large-scale operating system are not available. A free-standing implementation is only required to support the part of the Standard Library contained in the headers <stddef.h>, <stdarg.h>, <limits.h>, and <float.h>.

The Xtools eZ80Acclaim!™ compiler actually implements much more of the Standard Library than required by this definition, including the majority of the headers and functions of the Standard Library. These library modules are delivered with the compiler inside each distribution of ZDS II, in the form of both source code and pre-compiled libraries. For a full listing of the library functions supported by the compiler, see the *ZDS II User Manual -- eZ80Acclaim!*™ (UM0144).

Because the eZ80Acclaim!™ is an 8/16/24-bit processor, double-precision (64-bit) floating-point computations would be very slow on the eZ80Acclaim!™ and are not supported. The compiler treats the keyword double as being identical to float and implements single-precision IEEE standard floating-point values in either case. For performance reasons, the Xtools compiler does not implement the full IEEE floating-point standard: overflow/underflow detection and the NotANumber convention are not supported. These restrictions should not affect most embedded developers' use of floating-point computation.

Returning C structs from a function by value is expensive and slow, two penalties that should generally be avoided in the embedded world. The alternative of passing structs by pointer as function arguments, allowing the called function to modify them if desired, is much better programming practice. In the Xtools compiler, returning a struct by value is not allowed.

Guidelines for Writing Robust and Efficient Code

In addition to standard good practice for developing C applications in any environment, there are some additional issues that embedded developers need to concern themselves with, especially when trying to keep code size to a minimum. In this section we offer some advice on several topics that frequently cause problems in user applications: ANSI type promotions, the volatile keyword, large local arrays, use of the floating-point library, and the standard library function sprintf().

ANSI Promotions

To be ANSI compliant, the Xtools eZ80Acclaim!™ compiler must, of course, follow all the rules of the ANSI Standard in areas where the standard states that a given behavior is mandatory. Some of the standard's rules for integer type promotions require the compiler to generate code that is both much larger than necessary and almost certain not to work as intended. There are a couple of approaches to avoiding both of these problems. In this section we first explain the reason for this issue and then describe how to avoid it.

The standard has fairly elaborate rules for the promotion of integer types (elaborate partly because of C's convention that the actual sizes of the integer types can vary from one machine to another). The basic idea is that in every operation that takes two integer operands, the code should make sure that the two operands are of the same real type (i.e., occupy the same number of bytes) before proceeding when the operation takes place. To ensure this, if the types are different then the smaller type is "promoted" to the larger type before doing the operation. For example, if an 8-bit char is to be compared to a 32-bit long, the char will first be promoted by converting it to a long; then two longs are compared.

This promotion rule wreaks havoc in embedded applications most often because of another rule of the standard: the data type of any integer constant value is "int" unless prefixed with "U" (making it an unsigned int) or "L" (making it a long). Notice that there is no way to designate that an integer constant should be treated as a char. This means that in simple code like

```
char x;
x = 'T';
```

or

```
char y;
...
y &= 0x55;
```

the constants 'T' (i.e., the ASCII code for capital T) and 0x55 are to be treated as being of int type. Despite the almost irresistible tendency of the embedded programmer to think of these as "char constants", they are not, according to the standard.

The following example will illustrate the problems that result from this situation. Consider the code

```
char buffer[20];
...
if (buffer[0] == 0xff) do_something();
```

This code will cause two problems. First, unnecessary object code will be generated to promote the char value buffer[0] to an int so that it can be compared to the int value 0xff. But more surprisingly, the comparison will always say that the two values are unequal, even when the value of buffer[0] is 0xff! That happens because buffer, not being explicitly stated to be unsigned, is taken to be an array of signed chars. Therefore, when converting it to an int (which by default is 24 bits in the eZ80Acclaim!™), its value is sign-extended to 0xffffff. However, the constant is taken to be already a signed int, so written in the same format its value is 0x0000ff. This is virtually certain not to be the behavior intended by the developer, but it is correct compiler behavior and is required for compliance with the ANSI standard. The most widely used C compiler for the desktop behaves the same way. What's unique to the embedded environment is just the prevalence of code like this as designers, rightly, try to minimize the data sizes of their variables to reduce memory requirements.

Aside from the incorrectness issue (from the perspective of the code designer's intentions), the code bloat problem can easily be even worse than we have suggested so far. Consider code like:

```
char a, b, c;
...
a = (b | c) & 3;
```

Since 3 is an int in this expression, the compiler has to generate code to promote both b and c to ints, do all the operations on the right-hand side of the assignment on int (24-bit) quantities, and then convert the result back to a char at the end of the statement before assigning it to a.

Fortunately, there are two relatively easy ways to avoid all of these problems. The surest and most portable is to explicitly cast char constants to char:

```
char buffer[20];
...
if (buffer[0] == (char)0xff) do_something();
/* Now it works! */
```

This is the only way to create char constants while remaining in strict compliance with the standard. Since this uses only elements of the language standard, it is guaranteed to give the same results on any platform and with any compiler. It also forces the programmer to think about and explicitly specify the sizes of the constants used in his code, which is another step toward a cleaner, less ambiguous coding style.

The Xtools C compiler provides another way to get this result. The user can disable strict ANSI promotions by deselecting the check box for **Project > Settings > C > Code Generation > ANSI Promotions**. This will have the same effect of treating both sides of the comparison in the last example as (char) type. In the great majority of cases this should be safe to do. The one caveat of this approach is that the compiler then has no way of knowing about any exceptional cases in which the programmer really does want a constant – which could fit into a char, and is used with chars in an expression – to be treated as an int. This can sometimes cause problems if the constant is used as part of a complex expression as in the next example. As usual, the only sure solution is for the programmer to think carefully about data sizes in any cases where that's a critical concern.

In some cases, the compiler may be able to avoid the problems we have discussed here by optimizing away the unwanted promotions. However, it is more difficult than it would first appear to implement an optimization that does this as safely as is required. Compliance with the standard requires that even if the promotions are not actually done, the results in all cases (i.e., for all possible values of the operands) must be the same as if the promotion had been done. Clearly the safest way for the compiler to comply with this requirement is to actually do the promotions. The user's best approach to avoiding unwanted promotions is therefore to take one of the two approaches described above to make sure they aren't done.

On the other end of the spectrum, the absence of promotions when needed can also sometimes cause incorrect results. Consider this code, in an application where the size of an int has been defined to be 16 bits:

```
#define PROCESSOR_CLOCK_FREQ 18432000 /* 18.432 MHz */
#define UART_BIT_RATE 9600 /* 9.6 kbit/s */
#define BRDIV (PROCESSOR_CLOCK_FREQ / (UART_BIT_RATE * 16))
```

The calculation for BRDIV comes out completely wrong.

Here the culprit is a necessary promotion that does not take place. Again, both 9600 and 16 are taken by the compiler to be ints. But in the calculation of BRDIV, their product is too large to fit into a 16-bit quantity, and so is truncated, giving a completely incorrect result. (The constant 18432000 in this case was treated as a long, since the compiler can tell that it's too large to fit into a 16-bit int.) The promotion to a long doesn't occur until the next step in the complex calculation when this product has to be divided into a long, which is too late to save the situation. Again, the compiler behavior is correct but the result is wrong.

There are several ways to fix this problem. The safest is to promote the constants involved to longs (which by the rules of type conversion, will force any other values used in calculations with these constants also to be longs). So either or both of the following changes will solve the problem:

```
#define UART_BIT_RATE 9600L /* 9.6 kbit/s */
```

or

```
#define BRDIV (PROCESSOR_CLOCK_FREQ / UART_BIT_RATE * 16L))
```

Volatile

The keyword volatile was a relatively late addition to the ANSI standard, but is crucial in many embedded applications. Normally, a compiler assumes that the values of variables do not change except when the program writes to them explicitly. But this assumption can be disastrous in an embedded system where the variable represents the contents of a hardware register that can be modified asynchronously by the system hardware. As an example, consider code like:

```
int system_var = 0;

for (counter = 0; counter < 1000; counter++)
{
   if (system_var)
   {
      /*  ... critical processing loop ... */
      system_var = 0;
   }
}
```

Here the programmer's intent is that system_var, which is updated by hardware when certain system events take place, be used as a flag to drive critical processing when the events occur. However, the optimizing compiler can see that system_var is only assigned to in two statements, and is assigned to be 0 in both locations. Therefore, the compiler would normally be entitled to assume that system_var is 0 at all times. In this case, that means that the condition if (system_var) can never be true; therefore the critical processing loop can never be reached (it is "dead code"). Therefore, in turn, the entire contents of the for loop are empty and the compiler is justified in generating no object code for this at all! Here's a case where the compiler's reduction of code size is a bit too extreme for anyone's taste.

The solution is to declare system_var as volatile:

volatile int system_var = 0;

The volatile keyword was added to the language to handle exactly this situation. It lets the compiler know that this variable must be assumed to be unknown at all times, so that its value must be freshly read every time it is accessed.

Large Local Arrays

Due to the eZ80Acclaim!™ processor architecture, accesses to stack variables can be done efficiently as long as the stack offset is no larger than 127 bytes. For larger stack offsets, the variables must be accessed by a different method which is much less efficient in both code size and execution speed. This limitation doesn't often come into play unless the programmer allocates a local array that exceeds this size. Manipulating large arrays will be much more efficient if the arrays are allocated as global or static variables.

Floating-Point Library

Users who are working with floating-point values in their calculations need to make sure that they are linking to the "real" floating-point library. The Xtools distribution provides a dummy version of the floating-point library as well, as documented in the section "Troubleshooting C Programs" of the ZDS II User Manual. In that dummy version, all of the floating-point functions are replaced with stubbed-out versions, reducing code size to a minimum. If these stubbed-out versions are linked into an application that does actual floating-point calculations, garbage results will be computed.

To link in the real floating-point library, make sure you have checked the box **Project > Settings > C > General > Use Floating Point Library**. As with any library, the linker will pull in functions from the floating point library only if they are actually called by your application.

The reason for the existence of the dummy floating-point library is explained in the item on sprintf(() below.

Sprintf

One of the most common causes of user code becoming substantially larger than expected is the use of the standard library function sprintf(). This is, of course, commonly used in embedded applications for tasks like outputting text to a display device. Unfortunately, it typically increases the size of the overall application by something in the neighborhood of 5 kbytes, even if used only for a couple of simple calls.

The problem is that sprintf(), like all members of the printf() family of functions, must be prepared to accept a great number of formats and so the code for sprintf() contains calls to a large number of other functions. The ZDS II linker is smart and, when resolving symbols, will only link in code for functions that may be called by the application – it doesn't link in the entire library containing those functions. So if you check the box **Project > Settings > Linker > Input > Use C Runtime Library**, which allows the linker to link to the pre-compiled library if necessary to resolve function calls, it will pull in only those functions called by sprintf(), plus the functions called by those functions, etc. The trouble is that by the time all these calls are resolved, a large number of functions have been pulled in at a significant cost in code size. The basic difficulty here is that the linker can see the large number of functions that may be called by sprintf(), but doesn't know that in your application the number of functions that will be called may be much smaller if, for example, you only use one or two simple formats.

There are several things the user may be able to do to reduce this impact. Obviously, if your application has no need of sprintf() you will be better off avoiding it, but in many cases this isn't possible. The next most effective weapon is to disable the floating-point library, if you aren't doing any floating-point calculations. Users are often surprised to find functions from the floating-point library being linked to their application when the application doesn't use floating-point at all. The culprit is usually sprintf() or one of its relatives. Since sprintf() contains calls to (very large) functions for formatting floating-point values, the linker has to resolve those calls by linking those functions to your application at a large cost in code bloat. It is for exactly this reason that the Xtools distribution includes the stubbed-out version of the floating-point library, described above under the heading "Floating-Point Library". This will reduce the code size significantly. To link with this dummy version of the floating-point library, deselect the check box **Project > Settings > C > General > Use Floating Point Library**.

If code size is of critical concern, you may be able to get a significant savings by modifying the source code of the relevant modules in the standard library. This is somewhat laborious but may be worthwhile depending on circumstances. The module to focus on if you try this is uprint.c, which does most of the work for sprintf() and can be trimmed with some trial and error, removing code that supports unused formats.

Project Settings and Configuration

In this final section, we offer a few comments on project settings and configuration issues.

Optimization settings

The best combination of optimization settings can depend on the mix of code within a given project, so when trying to obtain the smallest code size or fastest execution, some experimentation is a good idea. The two main optimizations are available on the page **Project > Settings > C > General > Optimizations: Minimize Size and Maximize Speed**. Since in most cases smaller code also runs faster, in the great majority of C code these two optimizations will produce exactly the same object code, but there can be small differences. It is also possible that even the “Minimum Size” optimization can actually increase code size, by applying a trade-off that will cut code size in most applications but doesn’t work in your particular application. (One particular case that is known to cause this is the presence of switch statements with only 2 to 3 cases.) Use your map file to check code size results and, again, experiment.

For reasons described above in the section on ANSI Promotions, disabling the setting **Project > Settings > C > Code Generation > ANSI Promotions** will eliminate some type conversions that can, for some applications, result in a significant reduction in code size and execution time.

Another setting which usually, but not always, gives a modest decrease in code size is to select **Project > Settings > C > General > Debug Information > None**. When the compiler is asked to generate debug information, it also disables some optimizations that tend to save space but confuse the debugger.

Greater control over individual optimizations can be achieved through the selection **Project > Settings > C > Optimizations > Optimizations > Custom**. However, there is generally no reason to go to this level of granularity. The Xtools compiler applies all the optimizations that it safely can, consistent with the higher-level optimization settings.

Standard setup

The default boot module provided for eZ80Acclaim!™ projects sets up a number of required initializations. Users who for whatever reason choose not to use the default boot module need to understand the services provided by this module and create their own replacements if necessary. The default module, whose source code is included in the release, is usually a good place to start and a good example. The contents of this module are described in the FAQ in the ZDS II installation for each release. In a nutshell, this code sets up some necessary vector and jump tables, disables peripheral interrupts during startup, initializes the memory device arrangement, and then initializes the C runtime environment.

One item that is sometimes overlooked when users create custom boot modules is to set the symbol __heapbot appropriately. If any dynamic memory allocation is done in the user's application, malloc() will ultimately need to resolve this symbol so that it knows where to find the memory heap. Choosing an appropriate location for this depends on details of the user's memory map.

Header Files and Project Organization

The ZDS II distribution includes specific header files for each member of the eZ80Acclaim!™ family, which simplify the job of developing embedded C code for the individual family members (e.g., eZ80F91, eZ80L92, etc.). These header files define names and addresses (in the processor's internal I/O address space) of all the Special Function Registers (SFRs) of the given family member. When the appropriate header file is included in your project, you can access each SFR by name in your C code. The SFR names used are given in the Register Map section of the Product Specification for that family member, which is included among the documentation in the ZDS II installation.

It's not necessary to specify the variant-specific header file such as eZ80F91.h in your #include statements. Instead, you can simply say

```
# include <ez80.h>
```

When this file is included, it will automatically pull in only the single variant-specific header that befits your project. The compiler determines which header to use on the basis of the setting **Project > Settings > General > CPU**. Note that if you examine the "External Dependencies" in the project panel of the ZDS II IDE, you will see all of the variant-specific header files because the IDE doesn't take account of which #ifdef statements inside <ez80.h> evaluate to true or false. However, only one of these headers is really pulled into your project.

Since these headers are all located in the directory "include" below your ZDS II installation directory, that directory must be among the directories listed in **Project > Settings > C > Preprocessor >Include Paths**. You do not ordinarily need to do anything to set this up, as it should be included by default in the "User Paths" part of that dialog setting.

The Xtools toolchain places no special requirements on the directory structure you use to organize your project. You can use the **Project > Add Files** feature to browse to your source files wherever you choose to locate them. As with any software build system, you will need to make sure that if you change the locations of header files and object files from the defaults, you also update the relevant project settings so that the compiler and linker, respectively, can find them.

The one subtlety that can crop up occurs if you are using a fixed Link Control File rather than letting the system build a fresh one with each build to match your project settings. This happens if you have selected **Project > Settings > Linker > Input > Link Control File > Use Existing**. In this case, when you add a new file to your project and build it, the linker does not automatically become aware of the new object file and add it to the link. You will need to go in and edit the Link Control File to add the new object.

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