White Paper

Using the ZiLOG Xtools Z8 Encore!® C Compiler

WP000602-0904

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

Abstract

The Xtools Z8 Encore!® 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 Z8 Encore! 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 Z8 Encore! 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 Xtools Z8 Encore! compiler offers several C language extensions to enhance the language's support for embedded development and tailor it to the Z8 Encore! processor. Exhaustive technical detail on the comparison of the Xtools Z8 Encore! compiler to the ANSI Standard are given in the ZiLOG White Paper, *ZiLOG Z8 Encore! Compiler Compliance With ANSI STANDARD C*, WP0008.

The most important extensions in everyday use are those that support the practical concept of alternative memory models. The basic concept of the Z8 Encore! memory models is that there are a couple of different, commonly used ways to partition the Z8 Encore! register file between areas used for the stack, and for local and global variables. By giving the compiler information on where in the register file these different kinds of data will be located, the user allows it to choose the most efficient instructions and addressing modes for any given data access. The better model to use depends on the memory requirements of the user's application.

The two compiler-supported Z8 Encore! memory models are the small and large models, selected using the **Project > Settings > C > General >Memory Model** setting. Details of the models are given in the ZDS II User Manual. Briefly, in the small model the stack and, by default, all global and local variables are stored in the lowest 256-byte page of the register file, with addresses up to FFH. The advantage of this is that the instructions that manipulate data in this address range can use shorter addressing modes and so the overall code size will be significantly smaller. (Note that the space available for code is the same in both models; the "small" and "large" memory models refer to small or large default data memory spaces.) In the large model, the stack and, by default, global and local variables are stored in the extended register file with addresses beginning at 100H and extending to EFFH (or a smaller limit in parts that don't have 4K of register RAM). The memory spaces used by the small and large models are also called Rdata (00H-FFH, for "register" data) and Edata (100H-FFFH, for "extended" data), respectively.

The Xtools C compiler provides the extension keywords `near`, `far`, and `rom` to help users tailor their memory usage with finer granularity than provided by the memory models. These keywords are storage class specifiers that can be used to override the default memory allocations provided by the memory model. A `near` variable is always located in Rdata, a `far` variable is always located in Edata, and a `rom` variable is always located in Program rom, regardless of the memory model. These keywords are used like the standard storage class specifiers const and volatile. For example:

```
near char buffer[20];
```

If pointers to `near`, `far`, or `rom` variables are used, the pointer declarations must match the types of the variables to which they point (see the User Manual). This allows another potential code size optimization, since a `near` pointer occupies less space than a `far` pointer.

Another way in which the Xtools Z8 Encore! C compiler is adapted to the special needs of the embedded developer is its support for either static or dynamic call frames. The most familiar type of call frames, used exclusively by most desktop-oriented compilers, are dynamic frames: when a function is called, space is dynamically allocated on the stack to hold its return address, function parameters, and local variables. However, C also allows a more restrictive call-frame scheme using static frames. In this scheme, a single, statically allocated frame for each function in the program is allocated at compile time for storing the function's parameters and local variables.

The advantage of static frames is that since the compiler knows the absolute address of each function parameter, it can generate more compact code to access parameters than in dynamic frames where they must be accessed by offsetting from the stack pointer. For the Z8 Encore! instruction set architecture, this code size savings is substantial. It should be emphasized that the savings comes primarily not from using less space for frames, but from using less code to access data in the frames. Thus, it is primarily a savings in code space, not in data space. It could actually require more data space, although to mitigate this the Z8 Encore!

linker uses call-graph techniques to overlay some function frames that cannot be simultaneously active.

The disadvantages of static frames are that they do not support two features of the C language: recursion, and making calls through function pointers. To allow a broader range of applications to get the benefits of using static frames, the Xtools Z8 Encore! compiler provides the `reentrant` keyword as another C language extension. This keyword notifies the compiler that in an application that otherwise uses static frames, a dynamic frame must be used for any function declared `reentrant`:

```
reentrant recursive_fn (int x)
```

Obviously, if large numbers of functions in an application must be declared `reentrant`, the benefit of using static frames diminishes proportionately.

The Xtools compiler also extends the ANSI standard with 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 for Z8 Encore! User Manual* (UM0130) for details.

Another extension to the C standard is the ability to embed Z8 Encore! 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 for Z8 Encore! User Manual* (UM0130) describes two methods for embedding assembly code in a C file.

There are several areas in which the Xtools Z8 Encore! 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 Z8 Encore! 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 for Z8 Encore! User Manual* (UM0130).

Because of the Z8 Encore!'s nature as an 8-bit processor, double-precision (64-bit) floating-point computations would be very slow on the Z8 Encore! 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 fully supported. These restrictions should not affect most embedded developers' use of floating-point computation.

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, memory model and frame usage, the `volatile` keyword, large local arrays, use of the floating-point library, and the standard library function `sprintf()`. We close the section with some comments on how to benchmark code.

Users will also find the ZiLOG white paper *Z8 Encore! Reducing Compiled Code Size: Case Study and Programming Tips*, WP0009, useful as a case study of the application of many of these ideas to a real-world example.

Integer Type Promotions

The Xtools Z8 Encore! compiler follows all the rules of the ANSI Standard for evaluating integer expressions. However, some of the standard's rules for integer type promotions require the compiler to generate code that, in practice, is much larger than necessary. There are even cases where the standard-compliant code may not work as intended. There are a couple of approaches to avoiding 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 compiler should make sure that the two operands are of the same real type (i.e., occupy the same number of bytes, and are either both signed or both unsigned) before performing the operation. To ensure this, if the types are different, then the compiler must generate code to "promote" the smaller type 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 can wreak havoc in embedded applications because the standard defines the data type of any integer constant value as `int` unless appended 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 definition 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 16 bits in the Z8 Encore!), its value is sign-extended to `0xffff`. However, the constant `0xff` is by definition a signed `int`, so written in the same format its value is `0x00ff`. 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 the prevalence of code like this as designers, rightly, try to minimize the data sizes of their variables to reduce memory requirements.

Aside from missing the code designer's intent, 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;
```

Because 3 is defined by the standard to be an `int` constant, 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` (16-bit) quantities, and then convert the result back to a `char` at the end of the statement before assigning it to `a`.

Fortunately, it's relatively easy to avoid all these problems. The simplest way is to disable strict ANSI promotions when compiling the code. You can do that in the Xtools C compiler by deselecting the check box for for Project > Settings > C > Code > Generation > ANSI Promotions. This will have the effect of treating both sides of the comparison in the last example as `(char)` type. This is the best solution for most users who are developing typical embedded applications and, beginning with release 4.8 of ZDS II for the Z8 Encore!, is now the default. The Xtools Z8 Encore! compiler can provide a compile-time message when promotions are skipped by using the #pragma WARNSKIP. After encountering this #pragma, the compiler will issue a warning whenever it detects that a promotion will be ignored.

Applications that benefit from disabling promotions generally use `char` variables for 3 reasons: to represent 8-bit registers or hardware devices, to represent actual text characters like 'Q' in text-oriented applications, or to do simple arithmetic (for example, incrementing a counter, or adding very small integers). These are all characteristic of typical applications of the Z8 Encore!.

The kinds of applications that would have trouble with this approach are those that rely on the ANSI Standard's special handling of chars (and shorts) in some special circumstances. The ZDS II User Manual explains those circumstances in detail with example code. To summarize, these special circumstances occur either when a computation is done on unsigned `char` variables and then the result is assigned to a signed `int`, or when the comparison operators like < and > are used to compare a signed char to an unsigned char. Both of these are generally bad practice. In these special cases, the ANSI Standard promotions in effect put in a "miracle promotion" to "save" the bad code, by promoting the variables to a type that makes these dangerous operations give the expected result. However, the standard only does this for chars and shorts; it doesn't apply this rescue to the same dubious code if the variables are ints or longs. And the price for this rescue is bloated code for a great variety of `char` operations, including many that would be safe without it. It's almost always better to turn off the promotions and simply avoid writing that kind of code. Note that disabling promotions does means that the programmer must be sure that the value of the `char` variable will not exceed the limits of an 8-bit quantity.

If you can't purge your code of those kinds of operations, or if portability is a great concern, there is an alternative solution to the problem shown above. You can modify your code 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 compliant compiler. The drawbacks are the need to modify your code and to make sure that you catch every example of a place where the cast could be needed.

Finally, on the other end of the spectrum from what we have considered so far, the absence of promotions when needed can also sometimes cause incorrect results. Consider this code:

```
#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))
```

Memory Models and Call Frames

Several issues related to memory model and call frame usage can have a major effect on code size. The relative importance of these items and, in extreme cases, even the sign of the effect (i.e., whether they help or hurt) can vary widely depending on the mix of code constructs in your application, so when code size is an issue, it's a good idea to experiment.

In many cases the greatest single code savings is to use static frames if possible. As discussed above in "Embedded Modifications to the ANSI C Standard" on page 2, this option must be used with some care because errors will ensue if it is applied blindly and your code uses either recursion or calls through function pointers. You can avoid those errors by finding the functions that use those language features and declaring them `reentrant`. Note that in the case of function pointers, it is the functions to which the pointers refer, not the functions that make the calls, that must be marked as `reentrant`.

The small memory model will always produce more efficient code than the large model, if your application can use it. Use of the small model places stringent limits on the data space available for the stack and data, as discussed in "Embedded Modifications to the ANSI C Standard" on page 2. Use of the small model does not impose any restriction on your code size. It does help to produce smaller code, by enabling the compiler to use shorter instructions with more compact addressing modes. If you are near but slightly over the data-space limits of the small model, you might still be able to use the small model by declaring enough selected global or static variables as `far` to get your usage of Rdata down to the available space. The code used to access those `far` variables will be less efficient than the default data-access code, so if you follow this plan you should choose variables that are seldom accessed to be `far`.

Conversely, if (like most Z8 Encore! users) you are forced to use the large model because of your data space and stack requirements, you can still get some of the benefit of the more efficient code which is typical of the small model. To do so, carefully choose the most frequently used global or static variables and declare them `near`. This will help with both code size and even more so with execution speed, since your more frequently executed code will be more efficient.

One way of minimizing the amount of RAM data space your application needs is to allocate a single buffer in RAM to hold, for example, the largest of a number of strings you might need to display. The numerous strings are stored permanently in ROM where space is often less tight. Each string in turn is then copied from ROM to RAM at the moment when it is needed. Example code to do this is given in both the ZDS II distribution FAQ, and the *ZDS II for Z8 Encore! User Manual* (UM0130) under "Minimizing use of Rdata".

Another way of saving space when data space (Rdata and Edata; see the *ZDS II for Z8 Encore! User Manual*, UM0130) is at a premium would be to declare initialized tables that are not modified in the code with the `rom` keyword. The trade-off here is that the execution speed is likely to be somewhat slower, as the number of addressing modes available to the compiler for accessing `rom` variables is very small.

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 Z8 Encore! 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. If static frames are in use, this restriction does not apply.

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 for Z8 Encore! User Manual* (UM0130). 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.

Beginning with release 4.9 of ZDS II for the Z8 Encore!, the solution to these problems is to select the check box **Project>Settings>C>Code Generation>Generate Printfs Inline** (this check box is selected by default). When this option is selected, the compiler parses the format strings used in calls to `printf()` and `sprintf()`, and replaces the function call with direct calls to lower level helper functions. This removes the rather huge footprint required to parse the format strings at runtime, and results in only functions actually required by the format strings being linked into the application. For example, if you do not attempt to print a floating point value, a call to `printf()` no longer links in the floating point library.

In order for this option to be beneficial, it is important that all calls to `printf()` and `sprintf()` be replaced; otherwise you get the huge footprint of the runtime parsing routines plus the somewhat larger inline code for those calls that were handled inline. And in order that a call to `printf()` or `sprintf()` be replaced, it is required that the format be a string literal rather than a pointer to a character. For example, the following code will NOT allow the compiler to achieve the code size benefits:

```
char hello[]="Hello World\n";
printf(hello);    // Can't do this inline
```

but the following will work:

```
char hello[]="Hello World";
printf("%s\n", hello);    // OK
```

Another requirement for inline processing of `printf()` and `sprintf()` to be beneficial is that the related functions `vprintf()` and `vsprintf()` not be used. This is because these functions cannot be processed inline, so that using them pulls in the large footprint of the real (s)printf routines.

If either of these conditions is violated, the compiler will generate a warning message to tell you that the intended benefit of inline generation of printfs is not being obtained.

If you must disable the inline generation of `printf()` calls, and if your application does not use floating point calculations, you should disable the floating point library by deselecting the check box **Projects > Settings > C > General > Use floating Point Library**. This is because the print function does not know that you are not going to use a `%f` or `%e` format, and so must contain code to format floating point values, which will then generate calls to add, subtract, and compare floating point values, pulling in a large part of the floating point library for no purpose.

Benchmarking Code

Users often want to create a benchmark compilation of some test code for evaluation purposes. Even more importantly, most users are concerned with how they can be sure they have done the best job of optimizing the compiled code by some criterion, most frequently code size. Here we offer some concise guidelines on how to evaluate these kinds of questions.

The first point to make sure of in doing any kind of benchmark comparisons is to compare like against like. For the Z8 Encore!, this means that of course the only appropriate comparisons are to other 8-bit processors. If the benchmark code uses larger data sizes than 8-bit entities extensively, 8-bit processors in general will have to use more instructions than, say, a 16-bit processor to manipulate those larger entities and will therefore tend to have larger compiled code.

To generate the most compact or fastest code for any particular C program on the Encore, some experimentation is usually required. The first questions to look into are whether you can use static frames and/or the small memory model, which are usually preferable from the standpoint of efficiency. See "Memory Models and Call Frames" on page 8 for a discussion of the issues involved. Sometimes, minor changes to your code in this phase can have a big payoff if they allow you to use those configurations. Once you have settled on the memory model and frame type, investigate "Optimization Settings" on page 13. Again, a little experimentation will quickly show you which of the available optimizations are useful in your particular application.

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 tradeoff that will cut code size in most applications but doesn't work in your particular application. Use your map file to check code size results and, again, experiment.

For reasons described above in "Integer Type Promotions" on page 5, 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 startup modules provided for Z8 Encore! projects (startupl.asm or startups.asm for the large or small model, respectively) set up a number of required initializations. Users who for whatever reason choose not to use these default modules need to understand the services they provide and create their own replacements if necessary. The appropriate default module, whose source code is included in the release, is usually a good place to start and a good example. In a nutshell, this code sets up some necessary vector tables, initializes the stack pointer and register pointer, and then initializes the C runtime environment.

This initialization includes setting the near/far uninitialized global and static variables to zero, and copying the initialized variables from ROM to their near/far locations in the register file. If you are using the large model and you do not have any `near` data then you can modify startup to remove the unnecessary initialization, and vice versa for the small model. To do this you will have to add the modified startup module to your project and set **Project > Settings > Linker > Input > Startup Module > Included In Project**.

One item that is sometimes overlooked when users create custom setups is to set the symbol `_far_heapbot` appropriately. This only arises with the large model, because the small model doesn't have enough data space to perform dynamic memory management. In the large model, 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. If the user uses the default linker command file generated by the ZDS II IDE, `_far_heapbot` will be defined by the linker. Otherwise, choosing an appropriate location for this depends on details of the user's memory map. The heap is taken to begin at address `_far_heapbot + 1` and grow up from this location up to the stack pointer.

Header Files and Project Organization

The ZDS II distribution includes specific header files for each member of the Z8 Encore! family, which simplify the job of developing embedded C code for the individual family members (e.g., eZ8F6422, eZ8F3221, 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, as well as the Z8 Encore! interrupt vectors. 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 File Address Map section of the Product Specification for that family member, which is included among the documentation in the ZDS II installation.

Beginning with the version 4.7 maintenance release of ZDS II for the Z8 Encore!, it is no longer necessary to explicitly include a variant-specific header file such as ez8f3221.h in your projects. Instead, you can now simply say

```
# include <eZ8.h>
```

When this file is included, it will automatically define only those registers that are appropriate for your selected CPU variant. This is done using conditional compilation based on a preprocessor macro that represents your variant. This macro is set on the basis of the setting **Project > Settings > General > CPU** in your project. For backward compatibility, the older style, variant-specific header names will still be supported for a time, but those headers will simply point at the new, generic Z8 Encore! header eZ8.h.

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

Document Disclaimer