White Paper

Z8 Encore!® Reducing Compiled Code Size

Case Study and Programming Tips

WP000905-0308

Abstract

This White Paper presents a case study and programming tips for fitting the compiled size of an embedded application into a smaller memory space. A specific example is discussed in which a customer's application was reduced from 11 KB to 6980 bytes to fit the code memory of Zilog's Z8 Encore!® microcontroller.

Mission: Beat the 8K Limit

Have you ever faced a tough job of trying to get your compiled C code down below some stubborn size threshold? What would you think of an assignment to cut your code footprint by at least 29%—and you have 24 hours to do it? That's exactly the challenge that confronted us[1] recently. A Zilog Field Applications Engineer had asked for help from XTools in getting the Z8 Encore!® microcontroller qualified for use by a potential customer. The problem was that the largest Z8 Encore! part that the customer would consider was an 8K device, that is, 8192 bytes of code memory and the customer's code, which the FAE had already ported to the Z8 Encore!, compiled to 11,513 bytes. (The customer was previously running this code in a competitor's 16K part.)

The initial mission was defined as simply looking for any improvements to see if it might be worth expending more effort, but we soon found ourselves caught up in the excitement of trying to beat that 8192 byte limit in time for the next day's presentation. In the end, we managed to do more than that. We were able to cut the code size not 29%, but 39%—down to a total image of 6980 bytes. This report is the story of how we got there and, more importantly, how you can apply our methods to reduce your own code size. All the techniques we used are available to any user—you don't have to be a compiler expert, just willing to learn some of the things in your project that may be bloating your code unnecessarily and some of the techniques for avoiding them. Of course, your code may be tight already, and certainly we would not expect a 39% reduction to be typical, but if you have not considered these techniques, do you really know how much performance you might be leaving on the table?

In this report we will walk through the process we followed and show you our results in this real-world case study. One point we would like to emphasize is that, while you might be surprised how many areas we found where we could rewrite the source code and get a more compact executable, this is not just demo code but production code from a real-world embedded product that was already out in the marketplace. The code sizes we mention are real and so are the code snippets, though we have disguised them by changing variable names to protect the confidentiality of the real-world customer. We will explain many of the choices that can affect code size and show you how much space we saved by each change we made from the way this project first came to us. In some cases the customer or FAE had already made the best choice and we will show how much it would have cost to reverse those good decisions. We will also describe some choices that did not make

[1] The authors of this report are compiler developers in Zilog's Xtools group.

much difference for us but might be important for you if your application is different from this one.

General Approach

The techniques we used can be divided roughly into two approaches: changing compiler options, and modifying the actual source code. Selecting the compiler options involves setting the debug, code generation and optimization options, usually through the **Project Settings** dialog (though they can also be set by means of command-line options). Coding techniques involve changing the source code to influence how the compiler converts it into executable code. While we used both approaches in this project, we got most of our improvement from code modifications. The compiler settings were already fairly optimal by the time we were consulted.

Compiler options are simple to experiment with, but can have a big impact on the running application. If you can meet your code size goals using just the compiler options (as opposed to reworking your source code), that is the quickest route. Once you have settled on the compiler settings that seem to work best for your application, try to keep them unchanged if you then proceed to explore code modifications. The compiler options can strongly (and nonlinearly) affect how well a given code modification works. So if you get halfway through a project of modifying your code and then decide to change compiler settings, you might eliminate or even reverse the space savings you have already won—code tweaks that were helpful with the old settings could now be harmful. You really would need to re-examine all your code modification decisions from scratch if you did that.

The coding techniques we present here amount to ‘hand tuning’ your source code. They are usually only worth exploring after the code is fairly stable, since you do not want to spend time carefully optimizing a stretch of code and then throw it away next week. Also, it usually makes sense to take on this more labor-intensive approach only after determining that the best set of compiler settings still does not produce an image smaller than the limit you need to beat.

One important detail: how do you go about measuring just how big your code actually is? With the Zilog XTools compiler and linker, the code size of the executable can be found in the `.map` file. It is given as the size of the ROM address space in the section entitled, ‘SPACE ALLOCATION’.

In this project, we focused exclusively on the reduction of code size. How to reduce the amount of RAM used by an application is a topic for another day.

Selecting the Code Generation Options

When using the Zilog XTools C compiler, the choice of the compiler’s code generation options can have a significant influence on the size of the compiled image. Some of the effects you see will depend on the parameters of your application, but some general statements can be made about which choices are expected to yield smaller code size in most cases.

The options available for Z8 Encore!® compiler lie in 3 main dimensions, with a few minor choices as well. The major dimensions are Generate vs. Do Not Generate Debug Information, Large vs. Small Memory Model, and Static vs. Dynamic Frames. The minor dimensions are represented by the checkboxes on the Code Generation page under the C tab in the **Project Settings** dialog.[2]

The compiler settings will also influence the outcome of applying a particular hand-coding technique. For example, the function-call overhead using dynamic (function activation) frames is much greater than using static frames. We'll show below that in our case (using static frames), we got a size reduction from a technique of factoring similar blocks of code and placing them in a subroutine; but when using dynamic frames, this technique can potentially increase the code size.

All our experiments were run with optimizations in the compiler disabled. Our FAE had already determined that compiling without optimizations (for either size or speed) gave the best starting code size for this particular application. Optimizations are basically rules-of-thumb which are applied by the compiler to reduce code size or increase execution speed. The rules-of-thumb are based on assumptions which are usually but not always true. If the assumption is false, then the effect of the optimization can be the opposite of what was intended. The effect of one optimization based on a bad assumption can negate all of the gains due to other optimizations. That apparently was the case with the code we were given.

Generating Debug Information

When the C compiler is asked to include debugging information in the object code, it will also avoid some optimizations which would make debugging seemingly inconsistent. By enabling the inclusion of debugging information, you are accepting that your code size will be somewhat larger and execution times somewhat longer than in a version that has debug information disabled. In return for this you get code whose behavior will make more sense to you as you work on debugging it.

For example, the compiler sometimes loads the contents of a variable from a location in RAM into a working register. If the compiler is allowed to optimize for best performance, it might then perform operations on it corresponding to several source statements before storing it back to RAM. When the debugger displays the variable's value (for instance, in a watch window), it does so by looking at the 'permanent' location of the variable—in this case the RAM address. Therefore, to the user it would appear as though the variable were not being updated in the intervening statements, but would suddenly get the right value

[2] Though the original version of this white paper referred to now-obsolete compiler settings, use of those settings had no significant effect on resulting code size. The fact that the settings are now unavailable therefore has no bearing on the conclusions reached in this paper.

after the completion of the last statement. On the other hand, if the compiler is told to generate debug information it will avoid this confusing behavior by writing the variable back to RAM at the end of every source line. Turning OFF the debug information avoids the cost of all those unnecessary writes to RAM.

When the generation of debugging information is enabled, the compiler also generates additional code to:

- Set up a frame pointer on entry to each subroutine, and restore the previous frame pointer on exit.
- Use additional labels and jumps, so that tests appear to be evaluated at the beginning of a structured loop.

If code size is a concern, the **Generate Debug Info** option will normally be deselected.

In our example application, the **Generate Debug Info** Flag was selected in the project as it first came to us. By deselecting this and recompiling, the code size dropped from 11,513 to 11,102 bytes, saving us 411 of the 3321 bytes we needed to make our target. Our mission was underway.

Large and Small Memory Models

The Zilog® C compiler for the Z8 Encore!® supports large and small memory models. In the small memory model, all data including global and local variables plus the run-time stack must fit into the first page of RAM in the range 0x020 through 0x0FF. If these restrictions don't provide enough space for the application, global and static variables can be placed into locations 0x100 through 0xEFF in RAM using the `far` keyword explicitly, but the fact that the stack is implemented in page 0 of RAM is a restriction that cannot be gotten around in the small model.

When a variable is accessed, different code is produced depending on whether the large or small memory model is in effect (or equivalently, if the variable is declared to be `far` or `near`, respectively). Regardless of the model used, instructions used to move data to and from a working register will occupy the same number of bytes (3), but operations which manipulate data directly will be one byte larger if the data is `far` (as is the default in the large model). Therefore, the small memory model will usually produce smaller code than the large memory model. However, the small memory model is not appropriate for designs which use larger amounts of global and static data and stack space.

In our example application, the large memory model had to be selected because the total space required by the declared variables was already too large to fit into near memory, and no space would have been left for a run-time stack in the small model. Therefore, this optimization was not available to us, but is well worth using if your application will allow it.

Static vs. Dynamic Frames

A frame refers to the activation record produced by the C compiler as it compiles a function. The activation record holds the parameters and local variables associated with a function invocation. Since dynamic frames located on the stack are by far the most common way to implement this, the activation record is often familiarly referred to as the function's 'stack frame,' but there is an alternative method—the static frame—implemented by some compilers including the Zilog XTools Z8 Encore!® compiler.

When the dynamic frame model is used, frames are allocated on the stack at run time. The activation record for a function does not actually use any data memory until that function is invoked in the course of execution. If a function invokes itself (recursively) either directly or indirectly, it will require one frame for each copy of the function that is active.

One advantage of using dynamic frames is that it makes efficient use of data memory when local variables are used extensively. The memory allocated to local variables within a function invocation is freed up and may be reused after control is returned to its caller.

When the static frame model is selected, frames are allocated within static data memory by the linker. The frame for each function occupies memory regardless whether that function is actually invoked during execution. However, the linker improves memory usage for the frames by generating a call graph of the functions and reusing the memory for frames of functions whose activations are mutually exclusive.

Compiling your program with static frames will often reduce the code size, but there are a couple of features of C that static frames do not support or only support with restrictions: namely, recursion and function pointers. Therefore, if you are going to use static frames you must ensure that either these language features are not used in your program or that you take care to handle them correctly. Luckily, many embedded applications in fact do not use these features.

Each function will never have more than one frame in the static frame model, even if it is invoked recursively. Since this may lead to unexpected behavior, it deserves closer scrutiny. In this static frame model, each function has but one instance of its data. It is as if the `static` keyword has been applied to every parameter and local variable in every function in the program. A function written as

```
int sum(int a, int b)
{
  int c;
  c = a + b;
  return c;
}
```

is compiled as if it had read

```
int sum(static int a, static int b)
{
  static int c;
  c = a + b;
```

```
    return c;
  }
```

In other words, the parameters and local variables use the same location in RAM each time the function is called. (Using the `static` keyword in a parameter declaration is not allowed in standard C; we use that syntax here for illustration purposes only.)

Direct or indirect recursion involving functions with static memory or compiled using static frames will likely not work as expected. If static frames are selected, the way in which a design uses parameters and local variables should be examined carefully to ensure that it will still work. If it is determined that a function is truly recursive and that it requires dynamic frames, this can still be accomplished by affixing the Z8 Encore!® microcontroller's `reentrant` keyword to the function declaration. The compiler will then set up dynamic frames just for the function or functions designated as `reentrant`, while continuing to use static frames for all other functions.

A further restriction of using static frames is that they do not allow making function calls through function pointers. Again, this restriction can be overcome for individual functions by declaring them `reentrant`.

In the customer code we were working with, recursion and function pointers were not used, so we were free to use static frames if we so chose. As it turned out, in this program all data was declared globally (that is, there were no local variables). Therefore, the choice of global verses static frame allocation had no effect upon how variables were allocated. On the other hand, the allocation of parameters passed to subroutines is affected, and this did give us an improvement when we selected static frames.

When dynamic frames are in use, local variables must be accessed through the frame pointer, rather than through a static RAM address. In contrast to operations that can be performed directly on RAM locations in the static model, at least one operand must be loaded into a working register in the dynamic model, which costs more code space. In addition, using the dynamic model requires the establishment of a frame pointer when entering a subroutine and restoring the old one upon exit.

To demonstrate that the dynamic memory model was a bad choice in this case, we tried a test build with dynamic frames. The code size grew by 161 bytes, so we quickly reverted to static frames.

ANSI Promotions

Concerning arithmetic and comparison operators, the ANSI standard for C specifies that quantities with a lower precision (smaller size) than `int` are converted to `int` before the operation is performed. In most cases, this conversion has no effect on the result of the computation; it merely expands the size of the code unnecessarily. However, in expressions which combine signed values (`char` or `int`) with values having the type `unsigned char`, a different result will be obtained, depending on whether promotions are applied as specified. Generally code that relies on these ‘ANSI (type) promotions’ to work correctly is an example of one bad programming practice or another.

The project described in this white paper was developed using ZDS II v4.8 and the associated Z8 Encore!® compiler. At that time, Zilog® recommended turning OFF (deselecting) ANSI Promotions to allow that version of the compiler to generate more compact code. However, this practice runs the risk of generating code whose behavior does not conform to the ANSI standard. This problem can be avoided by careful coding, as it was in this particular project. However, not all users have been able to avoid problems, and it is clear that this option causes an unacceptable level of trouble in user applications.

To correct this issue, the option to disable ANSI promotions has been deprecated in ZDS II for Z8 Encore! v4.10.0. At the same time, the compiler has been significantly enhanced to generate more compact code and avoid truly unnecessary type promotions, while conforming to the ANSI standard. With these changes, the code size cost of retaining ANSI promotions is usually negligible and there is no need to use the deprecated option in new projects.

The project described in this white paper was originally built in ZDS II v4.8 with ANSI promotions disabled, and ANSI compliance was preserved by using explicit casts (from char to int or long) where necessary. At that time, we found that turning on ANSI promotions made the code grow by 852 bytes. In the ZDS II v4.10 compiler, ANSI promotions have a much smaller effect on code size, so we now recommend keeping ANSI promotions enabled so that explicit casts are not needed to produce ANSI-compliant code.

Strict ANSI Conformance

In the ANSI C Language Specification, it is stated that an object should be updated at most once during the evaluation of an expression [ISO/IEC 9899:1999, §6.5, ¶2]. However, in the Z8 Encore! architecture, it is more efficient to manipulate objects directly in memory rather than loading them into a working (temporary) register. Doing so may violate this aspect of the ANSI C standard, but would rarely result in a discernible difference in the behavior of the application.

Some kinds of memory (for example, memory-mapped I/O or control registers) perform an action when read or written. In manipulating a variable mapped to such a location, conformance to this aspect of the C standard would be important, since reading or writing such a location twice or more would have different observable behavior from reading or writing it once.

Another realm in which ANSI conformance would be important is when there are asynchronous updates of a variable which is shared among different execution threads. In that case, the setting of a bit might trigger some action each time it is done. Multiple updates with an expression could then cause spurious triggering. Outside of these special situations, however, it is generally safe to disable strict ANSI conformance.

With the application we worked on in this effort, we selected the checkbox for Strict ANSI Conformance. As a result, the code size grew by 38 bytes. Since we did not anticipate encountering either of the special situations listed above, we continued our experiments with Strict ANSI Conformance deselected (turned OFF).

Use Intrinsics

This option specifies whether the compiler can call intrinsic functions through a special interface. If it is deselected then the compiler must use the normal function-call interface, which often causes more inline code to be produced for each invocation. For minimal code size, this option should be selected.

In our case study, the Use Intrinsics checkbox was initially selected. The only imaginable reason for avoiding the use of intrinsics is if you intend to substitute your own implementations for some of the standard run-time library routines. Disabling the use of intrinsics would then provide the 'hooks' through which your custom library routine(s) could be invoked.

For a test, we deselected the 'Use Intrinsics' checkbox and recompiled. The code size remained the same. Since we expected the code size to increase, we hypothesized that none of the intrinsic C functions was called directly by this program. We then investigated and found that this was indeed the case.

To summarize our results from just changing compiler settings, we found that all of these except the generation of debug information were already set to their optimal values in our case. We were only able to recover 411 bytes from these experiments and still needed to get almost 3000 more. We were forced to turn to code modification techniques to see if we could ferret out some more elusive excess bytes.

Coding Techniques

It is a general programming principle that no code should be repeated. Anytime you find yourself writing the same idioms over and over, you have identified a candidate for optimization. This applies equally to the contents of conditionals as to sequences of statements.

Another place where optimization can be applied is in reducing the complexity of a calculation. This can be accomplished variously by using a simpler type, applying algebraic identities, or replacing an operation which occupies a lot of code space with an equivalent one that can be implemented more compactly (a technique called 'reducing the strength' of an operator). We explore these and other code size reduction techniques below, and illustrate them with examples.

Moving Variables to Near Memory

Significant code size reductions can still be obtained within the large memory model, provided global and static data that are frequently used are placed within the first page of RAM by using the `near` keyword explicitly.

In the large model, the compiler by default places every variable in far memory. The appearance of a `near` keyword in a declaration causes the compiler to place the variable in near memory (0x20–0xFF). It can then generate more compact code, since near memory can be accessed using 8-bit addressing in place of the 12-bit addressing required for accessing far memory. If the total size of the variables declared to be `near` exceeds the space available in the first page of RAM, the linker will report an error; so you can experiment freely with moving variables to near memory and let the linker tell you when you have gone too far.

The next step in our code size reduction was to place as many variables as possible into near memory. In the large memory model, the run-time stack is allocated in far memory. This fact allowed us to fit all of the variables into near memory.

When this was done, the ROM footprint of the program fell to 9700 bytes, a savings of an impressive 1402 bytes or 12% of the original size of the entire application. Since our example uses mostly global (static) memory, many of the generated instructions manipulate data directly in memory (as opposed to manipulating data in a working register). Instructions which manipulate near data in memory are one byte shorter than those that manipulate far data in memory. This may account for the size reduction obtained by placing all variables in near memory.

In this application, this was the largest single improvement we found. It also put us about halfway toward our goal. Now it began to seem possible that we might be able to get close to the desired 8K limit.

Using Fewer Arithmetic Types

The basic Z8 Encore!® architecture does not contain hardware multiply and divide instructions. Therefore, multiplication and division are performed by subroutines. Even in processors which contain hardware multiply and divide units, these will usually only directly handle certain data types (by ‘types’ here we mean, for example, 16-bit integers) and the multiplication or division of other types must still be handled in a subroutine. When linked in from the Run-Time Library, any such subroutine will expand the size of the program’s ROM image.

If your application requires such functions, there is no way around including at least one version. However, it is important to realize that there are different versions of these subroutines depending on the data types being handled. For example, there may be different routines for dividing two 16-bit integer (int) quantities, two 32-bit integer (long) quantities and two 4 byte floating-point (float or double) quantities.[3]

By selecting just one arithmetic type for calculations and making sure—through the use of explicit casts—that all calculations are done using that type, you can avoid having to link in several routines which perform the same operation on different types. If possible, using the integer size which is handled directly by a hardware multiply/divide unit will provide the greatest savings. If, like the Z8 Encore!®, the processor lacks a hardware multiply/ divide unit, the greatest savings will be gained if the smallest integer is chosen (subject to the precision requirements of the application). Since the Z8 Encore! is an 8-bit processor, manipulating types larger than 8 bits often forces the compiler to generate multiple instructions to handle the carries, etc.

Also, it's often the case that operations done in floating point can be converted to operations using long integers. Assuming that the operands are scaled appropriately, the substitution of integer arithmetic for floating-point arithmetic can be done without loss of precision. Since the Z8 Encore! also does not do floating-point calculations in hardware, library functions must be linked in to do all floating-point computations and these functions tend to be large. Therefore, if floating-point variables can be completely eliminated from your program, it often gives a sizable code reduction.

The customer code had only one variable declared to be a float. Any expressions involving this variable would necessarily be performed using the floating-point version of each subroutine required for arithmetic operations. Some expressions also contained floating-point constants, which would similarly require linking in those floating-point subroutines.

A close examination of our example code revealed that the floating-point operations could all be performed using long integers. A new value of the variable `rate` was being computed by dividing the measured count during some arbitrary period by the number of clock ticks in the same period, and then scaling this by the number of clock ticks per minute, as follows. In the initial code, only the rate was declared as `float`—presumably to retain precision in the result even if it was not scaled correctly.

```
extern volatile unsigned short count;
extern volatile unsigned short tick_count;
extern unsigned short ticks_per_minute;
unsigned short new_rate;
double rate;

rate = count;
rate = rate / tick_count;
new_rate = rate * ticks_per_minute;
```

Since `count`, `tick_count`, `ticks_per_minute`, and `new_rate` are all 16-bit unsigned integers (unsigned short), this suggests that the calculation could be done using unsigned long arithmetic. However, the order in which the calculations were done caused a loss of

[3] In our implementation, float and double have the same precision.

precision, so the original programmer compensated by performing the calculation in the floating-point domain.

By placing the multiplication first, we ensure that the intermediate result retains full precision, and that the final result is also accurate to within the discretization error (one count per period). Then, the computation can be performed using long integers, and will still retain the required accuracy.

```
extern volatile unsigned short count;
extern volatile unsigned short tick_count;
extern unsigned short ticks_per_minute;
unsigned long rate;
unsigned short new_rate;

rate = count;
rate = rate * ticks_per_minute;
new_rate = rate / tick_count;
```

After replacing the floating-point expressions with equivalent expressions using only long integers, the code size fell to 8885 bytes, a reduction of 815 bytes (7% of the original code size). Most of this reduction was due to the fact that we were no longer linking in the following floating-point routines (see Table 1):

Table 1. Floating-Point Subroutines Eliminated

Name	Size (bytes)
_fpdiv	119
_fpftol	99
_fpupop1	31
_fpupop2	31
_fpultof	34
_fpmul	86
_fppack	276
TOTAL	676

The remaining 139 bytes of the difference can be attributed to two other changes. Simplification of the C expressions involving floating point numbers eliminated some calls that would otherwise have been made to the corresponding long integer operator. The calls to the long-integer-to-floating-point conversion routine and its inverse also disappeared.

After eliminating all floating-point operations, it was noticed that the example code was also using every combination of signed and unsigned short and long multiply and divide (8 routines total). By forcing all multiplications and divisions to be done using long arithmetic, we were able to eliminate half of these routines from the image. This was done by using an explicit cast of short integers to long (and unsigned short integers to unsigned long, respectively) of at least one of the operands of a multiplication or division between two short integers. The resulting code size dropped another 167 bytes to 8718 bytes.

Note: *A good improvement was obtained even though we had to use the longest and least efficient data type, but at least we were only using a single type. By this point we felt we were knocking on the door of that magic 8192 byte limit—only 526 bytes to go.*

Chained Calculations

Where a temporary variable or register is used to hold an intermediate result, it is more efficient to perform the entire operation before writing it back, rather than performing the operation piecemeal. At every ‘sequence point,’ a C compiler is required to write the result back to a variable that has been modified. According to the C Standard, there is a sequence point associated with the semicolon at the end of an expression. (There is also a sequence point at every function call, and in other locations.) By chaining expressions, you reduce the number of times a result has to be written back to memory. At the same time, you free up the compiler to allocate registers more efficiently. Both these effects contribute to reducing the overall code size. For example, the compiler is allowed to generate more efficient code for

```
x = y = z + 2;
```

than for

```
y = z + 2;
x = y;
```

In the example, there were two main opportunities for reducing code size through chained operations. Decision logic in the code involving the logical AND operator (&&) had been factored to avoid a syntax error generated by recursive macro calls; we resolved the macro recursion problem and coalesced the factored conditional expressions. For example, one particularly complex factored conditional expression is shown below:

```
if (A)
  if (B)
    if (C)
      if (D)
        if (E)
          if (!F)
            if (!G)
              if (!H)
                if (I)
                  /* Do something interesting. */
```

We were able to coalesce this into one compound logical expression, as follows:

```
if (A &&
    B &&
    C &&
    D &&
    E &&
    !F &&
    !G &&
    !H &&
    I)
                  /* Do something interesting. */
```

Second, there were some numeric calculations which had been broken up for readability. Recall that for our rate calculation we had:

```
rate = count;
rate = rate / tick_count;
new_rate = rate * ticks_per_minute;
```

This was rewritten as

```
new_rate = count * ticks_per_minute / tick_count;
```

and was then found to require fewer bytes in the executable image.

After coalescing the conditionals, we recompiled and found a code size reduction of 950 bytes to 7768 bytes. By consolidating calculations that could be chained, we gained another 103 bytes, thus reducing the overall code size to 7683 bytes.

We had now crossed the critical threshold: we would move the customer's code from a 16K part into the desired 8K part. But we still had a few more tricks up our sleeves and we wanted to give the customer some headroom for future expansion, if possible. So we kept going with some more code modification techniques.

Factoring Repeated Code

In general, code which contains repeated sequences will be larger than code in which those sequences have been factored out as a function. This is true if the function has no parameters and no return value, and the function call overhead is small. A project like this one in which variables are declared to be (filescope) static or global will tend to contain a minimum of parameter passing. Also, compiling using static frames will minimize the function call overhead. These characteristics of the code in the customer's application made the approach of factoring out repeated code into functions particularly appealing.

In the code example, there were 21 repetitions of code as shown below:

```
  // Turn off device.
  if (device_active_low)
    device = 1;
  else
    device = 0;
```

and 9 additional instances that looked like

```
  // Turn on device.
  if (device_active_low)
    device = 0;
  else
    device = 1;
```

Replacing these with calls to functions to perform the decision and assignment saved a total of 325 bytes, and reduced the overall code size to 7358 bytes.

Each instance of the first repeated sequence was replaced by the function call `device_off()`; each instance of the second sequence was replaced by `device_on()`. We then moved one instance of each repeated sequence into their respective functions:

```
void device_off()
{
  // Turn off device.
  if (device_active_low)
    device = 1;
  else
    device = 0;
}

void device_on()
{
  // Turn on device.
  if (device_active_low)
    device = 0;
  else
    device = 1;
}
```

We also noticed that a list of assignment statements was being repeated when any one of several conditions caused the device to go from an active to a passive state. We factored this sequence of statements as a new function called `drop()`. Another, larger segment of code appeared to be intended to Flash an LED. This segment appeared in two places. By replacing the repeated (inline) segments with calls to `flash_led()` and `drop()` as appropriate, we saved another 205 bytes and drove the code size down to 7162 bytes. All together, by factoring repeated code segments into functions we managed to save 530 bytes, or 4.6% of the original application size.

Applying Logical Identities

There was a repeated section of code which looked unnecessarily cumbersome. The visual indicator for this is if there is a logical expression with many terms and multiple appearances of the variables involved. In this case, the code looked like:

```
  if ((!device && device_active_low) || (device && 
!device_active_low))

    device_off();

  else

    device_on();
```

Closer inspection revealed that this code segment will turn the device in question on if it is not currently ON, and OFF if it is currently ON. That is, it toggles the state of the device. This can be accomplished much more efficiently by replacing that entire code block with

```
    device ^= 1;
```

There was also a block of code which first tested whether a parameter was positive or negative, and then performed further tests in the separate domains. But since the contained tests did not depend on the domain, the outer test could be eliminated. Thus, the code

```
  if(TestVal  < 0 ){                      //Check for max value
    if((-TestVal)  > 0x0200){
      TestVal = -512;
    }//if
  }//if
  else{
    if(TestVal  > 0x0200){               //Check for max value
      TestVal = 512;
    }//if
  }//else
```

could be rewritten as

```
  if(TestVal > 512) TestVal = 512;
  if (TestVal < -512) TestVal = -512;
```

In the example code, these substitutions trimmed OFF a total of 75 bytes, leaving the net code size at 7078 bytes. This was not a dramatic improvement in our case, but you may be able to find more of these kinds of opportunities in your code—especially if you are already familiar with it, and have more than a day to spend.

Factoring Conditionals

The next step concerned the factoring of conditionals, so that common parts of the conditional tests did not have to be repeated. For example, if you have

```
if       (A && B && C && D) { /* Do something. */ }
else if  (A && B && C && !D) { /* Do something else. */ }
else                         { /* Do a third thing. */ }
```

the conditional can be factored to yield:

```
if (A && B && C)
{
  if (D) {/* Do something. */ }
  else { /* Do something else. */ }
}
else { /* Do a third thing. */ }
```

By applying this transformation in a few places, we were able to reduce the code size by another 51 bytes. The resulting image size came in at 7027 bytes. It would have been possible to apply this transformation in more places to achieve greater gains, but it was labor-intensive, and therefore susceptible to errors. As with the logical identities, we probably left some potential gains on the table because of our very short time horizon and lack of familiarity with the code. You may be able to do better!

Removing Redundant Initialization

In our implementation, the C compiler creates initialized global and static variables by allocating the initial value in ROM and then copying those values into data memory during startup. Therefore, all statically-initialized variables—even those explicitly initialized to zero—will consume an amount of ROM space equal to their size.

Explicit initialization to zero is never required, since the C compiler is required to initialize all static (and global) variables to zero in its startup code. There was only one instance of explicit initialization to zero in our example code. But by removing it, we were able to save 4 bytes of ROM space and push the code size down to 7023 bytes. Obviously, a more extensive use of explicit zero-initializers in your application could have a significant effect on the resulting image size.

Another optimization is available if the application explicitly initializes every variable before using it. In that case, the portion of the startup code which initializes data memory (either to zero or by copying the initial values from ROM) can be removed from the Zilog® standard startup module `startupX.asm` (where X is ‘l’ if you are using the large memory model and ‘s’ if using the small memory model). This special modification saved us another 43 bytes, thus reducing our total code size to 6980 bytes. This was the final version of the project that we delivered back to the Zilog FAE.

Note: *It could be dangerous to perform both of the above optimizations simultaneously. It is considered good programming practice to initialize all your variables explicitly. If, in our application, the one variable that was explicitly initialized to zero really needed to be initialized in that way, it would cost us only 4 bytes to restore that explicit initialization. Then, we can safely remove just the portion of the startup code that zero-initializes static data memory. In that case, our net code size would be 6984 bytes.*

Summary of Results

The results of our code size reduction effort are shown in Table 2. The most dramatic reductions in code size were due to moving variables to near memory (equivalent to using the small memory model), consolidating conditionals and eliminating floating-point computations. Significant code size reductions were also obtained by removing debug information from the image and factoring repeated code segments as subroutines.

Other techniques were applied, but yielded only modest size reductions. That greater gains were not obtained is due to characteristics specific to this customer’s application. Depending on your application, these techniques may still be valuable in reducing your code size.

Table 2. Sample Code Size Evolution (ZDS II v4.8)

Action	Size (bytes)	Difference (bytes)	Difference (%)
Original	11513		
Eliminated debug information	11102	-411	-3.6%
(Used dynamic frames)	11263	+161	+1.4%
(Enabled ANSI Promotions)	11954	+852	+7.4%
(Enabled Strict ANSI Conformance)	11140	+38	+0.3%
Moved variables to near memory	9700	-1402	-12.1%
Eliminated floating-point subroutines	8885	-815	-7.1%
Eliminated word multiply and divide	8718	-167	-1.5%
Consolidated conditionals	7768	-950	-8.2%

Table 2. Sample Code Size Evolution (ZDS II v4.8) (Continued)

Chained expressions	7683	-103	-0.9%
Factored out `device_on()` and `device_off()`	7358	-325	-2.8%
Factored out `drop()` and `led_blink()`	7153	-205	-1.8%
Applied logical identities	7078	-75	-0.6%
Factored conditionals	7027	-51	-0.5%
Removed redundant initialization	6980	-47	-0.4%
TOTALS	6980	-4533	-39.4%

Note: *Configurations shown in parentheses were discarded.*

Final Notes

We have reviewed a number of techniques which can be applied to reduce the code size of an image. We have shown that by applying these techniques to the code for a real world application, we managed to reduce the image size from 11,513 to 6980 bytes. The final code was only 60.6% of the size of the original code. 90% of this reduction came through rewriting the C source code in ways that allowed the compiler to generate more efficient machine code. This demonstrates that these techniques are effective at reducing code size, and that significant improvements are possible.

Note: *We went from being more than 3000 bytes away from even fitting into an 8K part, to having a reasonable degree of headroom: the user's application could now grow 17% larger and it will still fit into the ROM space in that part. And if you're wondering, the customer agreed that Z8 Encore!® 8K part would meet their technical needs and proceeded to the next round of negotiations.*

If there is a single moral to this little success story, it's this: if you are having code size problems, try looking to your own source code first. There just might be a leaner program ready to emerge from it—with a little help from you.

⚠ **Warning:** DO NOT USE IN LIFE SUPPORT

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.

Z8 Encore! is a registered trademark of Zilog, Inc. All other product or service names are the property of their respective owners.