Migrating from Assembly to C for 8-bit Microcontrollers

Techniques for Programming an 8-bit Microcontroller in C and Optimizing Code Size

by
Taylor Cox
ATMEL Corporation

#442 at the electronicaUSA Embedded Systems Conference
San Francisco, California 2004

Introduction

Assembly language has typically been the programming language of choice for embedded system programmers. Looking into the 8-bit microcontroller offerings from different vendors, one finds that these microcontrollers can be programmed using the high-level C programming language as well as assembly. Some microcontrollers have even been designed with high-level languages in mind as a means of programming, thus alleviating common high-level language bottlenecks. The embedded source code examples found in this paper will be based on ATMEL Corporation's 8-bit AVR microcontroller, which was designed for programming using the high-level C language. In order to program microcontrollers using the C language a *compiler* is required to translate the high-level language in to low-level machine instructions the microcontroller can execute. Utilizing a high-level language when programming microcontrollers offers key advantages over low-level languages while introducing a single 'gotcha' that can be overcome with careful attention and planning.

Assembly versus C: The Line of Compromise

As embedded projects get bigger and bigger, it becomes more complicated to keep track of everything done in a project in assembly (i.e. register usage). An increasing amount of time is spent on these housekeeping issues. Working in a high-level language, will allow the compiler to handle a lot of this work allowing the embedded programmer more time to do actual work and develop their embedded program.

A small project may be done quickly in assembly, but a large project will take considerable amount of development time. If one takes the same project and writes the embedded source using a high-level language the small project would take approximately the same amount of time to complete. However, the large project would take at least twice as long to complete in assembly than it would if a high-level language were used. It's a fact that the source code of an assembly program is three to five times larger than the same source code written in C.

Benefits and Consequences of High-Level Languages

The first major benefit of a high-level language is *portability*. Using a high-level language allows the embedded programmer to remove the specifics about the target architecture and focus on a functional implementation rather than a symbolic or hardware specific implementation. This allows for the embedded source to be re-targeted to another architecture with little or no changes to the actual source code.

Readability is the second advantage. A high-level language allows for easier reading (and writing) because the embedded source resembles that of written languages. The symbols and instructions appear in the programmer's native tongue rather than computer language. Accompanying readability is *maintainability*. Programs are easier to maintain when written in a high-level language because they easier to read.

The final major advantage is *modularity*. Modularity allows the embedded engineer to collect one's embedded source into modules. These modules can then be included into

various projects. Creating modules allows for *reusability*, or the re-using of source code in different projects. Utilizing the modularity of high-level languages one can significantly decrease a project's development time assuming that one is in possession of an embedded code library which contains rudimentary functions (i.e. Accessing the Serial Peripheral Interface).

Despite the advantages of programming using a high-level language, there does exist a consequence concerning the effectiveness of the compiler. Programming using a high-level language can result in reduced code *efficiency,* as the compiler may not translate ones high-level source to the optimum machine code. This consequence can be easily overcome or minimized by selecting a microcontroller which was designed to be programmed using a high-level language and by writing ones high-level source properly. Both topics will be discussed, the later being the primary focus of this paper.

Variable Data Types

Declaring a variable in assembly requires the embedded system programmer to allocate the register(s) necessary for variable storage. In systems with more variables than registers, one first needs to determine the lifetime of the variable. If the lifetime of the variable is finite then properly managing the resources should not be a difficult task. However, if the lifetime of the variable is infinite and the variables register is needed temporarily for another variable or task, then the register contents will need to be stored to memory and restored from memory once the register is no longer needed. Another benefit of the C programming language is in the handling of data types. The C compiler takes your variable declarations and generates the necessary code and allocation schemes for these variables.

The embedded designer must be careful as to the data types used to declare ones variables, as the C language specification does not control the sizes of data types. When considering 8-bit microcontrollers, typically a "char" is 8-bits, a "short" and "int" are 16-bits, and a "long" is 32-bits. It is strongly recommended to avoid using 16- and 32-bit variables as 8-bit microcontrollers typically have limited (if any) support for 16-bit operations, let alone 32-bit operations. Utilizing "long" data types in your embedded source will adversely affect the performance of your embedded application by decreasing performance and increasing your code size footprint in comparison to implementations using "int" data types.

Since our target is an 8-bit microcontroller, the use of 16- and 32-bit variables should not be used except under necessity. The following two examples details the code size comparisons for a simple counter loop for both 8- and 16-bit variables in C and their corresponding assembly code. These code snippets were taken from the C compiler LST output file, which details the translations from the high-level language to low-level machine language. The compiler's optimization level was disabled. The comments were added for the readers' benefit.

Example 1: 8-bit Counter Loop

```
unsigned char count8 = 5;     /* Declare a Variable, Assign a Value */
//         LDI   R16, 5             ; Initialize Variable
```

```
do                              /* Start Loop */
{
} while (--count8);             /* Decrement Counter & Check if Zero */
// ?0004:  DEC   R16                  ; Decrement R16
//         BRNE  ?0004                ; Branch if Not Equal
```

Example 2: 16-bit Counter Loop

```
unsigned char count16 = 5;      /* Declare a Variable, Assign a Value */
//         LDI   R24, LOW(5)         ; Initialize Variable, Low Byte
//         LDI   R25, 0              ; Initialize Variable, High Byte

do                              /* Start Loop */
{
} while (--count16);            /* Decrement Counter & Check if Zero */
// ?0004:  SBIW  R24, LWRD(1)         ; Subtract 16-bit Value
//         BRNE  ?0004                ; Branch if Not Equal
```

Carefully examining the assembly outputs of the C compiler, one notices the differences between the 8- and 16-bit counters and the resources used. One finds that only one register `R16` is used in the 8-bit implementation, while two registers `R24` and `R25` are used for 16-bits. One also shall find that the pre-decrement is implemented differently. The pre-decrement is a simple `DEC` instruction for the 8-bit implementation, while for 16-bit the `SBIW` (Subtract Immediate from Word) is used – a dedicated instruction for 16-bit data handling built into the AVR microcontroller!

The extra register in Example 2 has negative consequences. Example 1 requires three words (or six bytes) of code space, while Example 2 requires four words. Examining the performance of both examples, Example 2 requires six more clock cycles, as the `SBIW` is a two-cycle instruction while `DEC` is a single cycle instruction, plus the extra single cycle `LDI` instruction required to initialize the second register. Clearly, it is advisable to use the smallest data type possible to meet your needs; you will receive reduced code size and increased performance as a result.

One should be aware that the C programming language does not offer a “bit” data type, a common need for embedded programs. As a result, if a single bit variable is needed the embedded designer is advised to use the 8-bit “char” data type. Although microcontrollers do not support bit variables they do support bit manipulation, a topic addressed later on in this paper.

Global versus Local Variables

Just as different variable data types exist, different types of variables also exist. The embedded designer is left the task of declaring the variable and should consider the variables’ function to determine whether it should be global or local. A *local* variable is a variable that is used only inside a single function or routine and should be declared within that function. A *global* variable on the other hand is utilized in more that one

function and/or file and should be declared outside the function block. See Example 3 for further details.

Local variables are typically assigned to the microcontroller's register file. A register is allocated for the variables use until the variable is no longer referenced or the end of the function has been reached.

A global variable resides in the microcontroller's data memory. A location in the microcontroller's SRAM is reserved for the global variable use and cannot be used by other resources. Prior to modification or use the global variable must first be loaded into a scratch register, once modified it is necessary to update the global variable's SRAM location with its new contents.

Example 3: Global and Local Variables

```
char global;                  /* This is a Global Variable */

__C_task void main (void)
{
      char local;             /* This is a Local Variable */

      global -= 45;           /* Subtraction with Global Variable */
//         LDS   R16, LWRD(global)  ; Load Variable from SRAM to
                                    ;   Register R16
//         SBIW  R16, LOW(45)       ; Perform Subtraction
//         STS   LWRD(global), R16  ; Store Modified Data to SRAM

      local -= 34;            /* Subtraction with Local Variable */
//         SUBI  R16, LOW(34)       ; Perform Subtraction Directly on
                                    ;   Local Variable in Register R16
}
```

Both the LDS and STS (Load and Store Direct to SRAM) instructions presented above in Example 3 are two word and two-cycle instructions. Operating on the global variable requires ten bytes of code space and five clock cycles for execution while the local variable requires only two bytes and a single clock cycle. Clearly it is more advantageous to declare variables locally whenever possible as one will have a much more efficient embedded source.

A local variable whose contents must be retained between multiple calls of the function can be declared using the keyword "static". If a local variable is declared as static, it is loaded into a scratch register and stored back to SRAM at the beginning and end of the function respectively. Compared to global variables, static variables will provide optimized code if the variable is accessed more than once within the subroutine.

If global variables are absolutely necessary within your embedded source, there are a few hints, which may either improve performance or reduce the number of required global variables. Reducing the number of global variables may be possible by utilizing the passing of function parameters (in registers) rather then declaring multiple global variables. A performance increase may also be possible when using global variables, by

loading the variable in to a register and manipulating the register rather than subsequently accessing the global variable from within the same function.

Optimization of Global Variables

An easy way of optimizing global variables can be achieved by collecting them into structures whenever possible. A *structure* (or record) is a grouping of one or more variables, including those of different data types, combined together under a single name for convenient data handling. Grouping the global variables in structures allows the compiler to generate code, which accesses these variables indirectly via pointers rather than directly addressing memory locations. See Example 4.

Example 4: Collecting Global Variable in Structures

```
typdef struct
{
      char sec;
} t;

t global;                    /* Declare a Global Structure */

char min;                    /* Declare a Global Variable */


__C_task void main (void)
{
      t *time = &global;     /* Pointer to a Global Structure */
//        LDI    R30, LOW(global)   ; Initialize Z Pointer, Low Byte
//        LDI    R31, (global >> 8) ; Initialize Z Pointer, High Byte

      if (++time->sec == 60)  /* If Variable sec Equals 60 */
      {
//        LDD    R16, Z+2           ; Load with Displacement
//        INC    R16                ; Increment
//        STD    Z+2, R16           ; Store with Displacement
//        CPI    R16, LOW(60)       ; Compare
//        BRNE   ?0005              ; Branch if Not Equal
      }
//  ?0005:

      if (++min == 60)        /* If Variable min Equals 60 */
      {
//        LDS    R16, LWRD(min)     ; Load Direct from SRAM
//        INC    R16                ; Increment
//        STS    LWRD(min), R16     ; Store Direct to SRAM
//        CPI    R16, LOW(60)       ; Compare
//        BRNE   ?0006              ; Branch if Not Equal
      }
//  ?0006:
}
```

The Z-pointer, one of three 16-bit memory pointers in the AVR is comprised of registers R31:R30 and is used along with the LDD and STD (Load and Store with Displacement) instructions to access the data memory when accessing global variables via structures. As in Example 3, the LDS and STS instructions are used when operating on the global variable "min". In the above example, the structured approach required only ten bytes of code while the global approach required fourteen bytes. It must be noted that the structured approach code benchmark doesn't include the required four bytes to initialize the Z-pointer. Adding the four bytes to initialize the Z-pointer and then comparing the two approaches yields the same fourteen byte outcome for this example, however if more than one variable resides within the structure then it would be more efficient to group the global variables inside structures as in the example presented above.

Accessing Single Bits

It is typical of embedded applications to need to address specific bits of an I/O register or single bit of a software flag register. There are two different means of manipulating bits in the C programming language both will be discussed, as well as their benefits and consequences. The first approach that will be discussed is Bit Masking and is demonstrated in Example 5.

Example 5: Bit Manipulation via Masking

```
/* Macros Definition */
#define BIT(x)            (1 << (x))      /* Bit Position */

SFR_B (PORTC, 0x15);                      /* PORTC Data Register */

__C_task void main (void)
{
      PORTC |=  BIT(0);       /* Set 0th Bit Equal to '1' */
//        SBI    0x15, 0x00         ; Set Bit in I/O Register

      PORTC &= ~BIT(0);       /* Set 0th Bit Equal to '0' */
//        CBI    0x15, 0x00         ; Clear Bit in I/O Register

      PORTC ^=  BIT(0);       /* Toggle the 0th Bit */
//        LDI    R16, 1             ; Load Immediate to R16
//        IN     R17, 0x15          ; Read I/O Register into R17
//        EOR    R17, R16           ; Exclusive OR Registers R16 & R17
//        OUT    0x15, R17          ; Output R17 to I/O Register

      if ( PORTC & BIT(0) )   /* Test 0th Bit, Is 0th Bit Set? */
//        SBIS   0x15, 0x00         ; Skip if Bit Set in I/O Register
//        RJMP   ?0001              ; Relative Jump
      {
            /* Insert Instructions Here */
      }
//  ?0001:
}
```

Masking provides efficient output from the high-level compiler, notice the single SBI and CBI instructions used for setting and clearing bits. The above bit-mask approach is guaranteed to work under all C compilers and the user has complete control over all bits within a byte. Bit-masks may be slightly more confusing to read but this can be overcome by implementing simple pre-processor macros for common functions as demonstrated below. Macros will be discussed in more depth as a topic later on in this paper.

```
#define SETBIT(x,y)     (x |=  (y))     /* Set Bit y in Byte x */
#define CLEARBIT(x,y)   (x &= ~(y))     /* Clear Bit y in Byte x */
#define CHECKBIT(x,y)   (x &   (y))     /* Check Bit y in Byte x */
```

A second approach to accessing bits is using a bit-field structure. A *structure* is a group of differently typed variables. Bit-fields can be implemented for both I/O and SRAM data memory. In a bit-field, as seen in Example 6, when accessing an I/O location one needs to treat the I/O location as data memory. After looking at the bit-field example, one can see that this approach is much clearer and easier to read and understand.

Example 6: Bit Manipulation via Bit Fields

```
typedef struct
{
      unsigned BIT0 : 1,
               BIT1 : 1,
               BIT2 : 1,
               BIT3 : 1,
               BIT4 : 1,
               BIT5 : 1,
               BIT6 : 1,
               BIT7 : 1
} IOREG;

#define PORTC    (* (IOREG *) 0x35) /* Locate PORTC in I/O Memory */


__C_task void main (void)
{
      PORTC.BIT0 = 1;         /* Set 0th Bit Equal to '1' */
//         LDI   R30, LOW(53)       ; Initialize Z Pointer, Low Byte
//         LDI   R31, (53) >> 8     ; Initialize Z Pointer, High Byte
//         LD    R16, Z             ; Load R16 with SRAM Location Z
//         ORI   R16, 0x01          ; OR Immediate with R16
//         ST    Z, R16             ; Store R16 to SRAM Location Z

      PORTC.BIT0 = 0;          /* Set 0th Bit Equal to '0' */
//         LDI   R30, LOW(53)       ; Initialize Z Pointer, Low Byte
//         LDI   R31, (53) >> 8     ; Initialize Z Pointer, High Byte
//         LD    R16, Z             ; Load R16 with SRAM Location Z
//         ANDI  R16, 0xFE          ; AND Immediate with R16
//         ST    Z, R16             ; Store R16 to SRAM Location Z

      if ( PORTC.BIT0 )        /* Test 0th Bit, Is 0th Bit Set? */
//         LDI   R30, LOW(53)       ; Initialize Z Pointer, Low Byte
```

```
//         LDI   R31, (53) >> 8     ; Initialize Z Pointer, High Byte
//         LD    R16, Z             ; Load R16 with SRAM Location Z
//         SBRS   R16, 0x00         ; Skip if Bit Set Register Set
//         RJMP   ?0001             ; Relative Jump
      {
            /* Insert Instructions Here */
      }
//  ?0001:
}
```

One should note that the same operations are performed in Examples 5 and 6, just using a different method for bit addressing. The bit-mask approach is clearly more code efficient then the bit-field approach, not to mention the verbose overhead required for using the LD and ST instructions also yields a performance boost. A problem still does exist for the bit-field approach.

The C Standard does not specify the allocation order of the bit-field. In other words the allocation of the bits (i.e. left to right or right to left) is compiler dependent and can vary from compiler to compiler. This proves to be a problem when portability is a concern. If one is using bit-fields, it is recommended to read the compiler manual to determine the allocation order of bits for proper I/O Memory alignment. Going one step further, the C Standard only dictates that only “int” and “unsigned” are valid data types for bit-fields. Some compilers, like the example above, have extended the specification to include “unsigned char” and allocate only one byte for the aforementioned structure. Compilers that have not extended the specification to include “unsigned char” and allocate two bytes, the structure implementation will not work.

As an aside, the topic of Endianness, or the storage order of a value larger than 8-bits in memory, deserves discussion. Two types of microcontrollers exist, Big Endian and Little Endian. Big endian processors store the Most Significant Byte (MSB) first followed by the Least Significant Byte (LSB) while little endian processors store the LSB first followed by the MSB. One can clearly see that if a value is stored using one endian and read using the other endian the data read will be incorrect. The C Standard does not specify endianness, but the environments for the same microcontroller architecture would be the same making endianness moot. However, if the embedded designer rolls their own code and then ports the application to another architecture endianness could come into play.

Functions versus Macros

A *function* can be thought of as a means of encapsulating some computation or algorithm, allowing an embedded engineer to utilize the routine without worrying about its implementation. Functions are equivalent to subroutines in assembly. Functions are typically composed of two parts – the Function Prototype and Function Body. The prototype is simply a declaration outlining the return type and parameters while the body contains the actual implementation of the algorithm and/or computation. See Example 7.

```
/*** Function Prototype ***/
return-type function-name ( parameter declarations [if any] );
```

```
/*** Function Body ***/
return-type function-name ( parameter declarations [if any] )
{
      /* Declarations & Statements */
}
```

A *macro* is simply a substitution, after the definition subsequent occurrences of the token name will simply be replaced by the replacement text. Macros can also be constructed to accept arguments and evaluate simple expressions. A template of a macro definition is shown below. See Example 7.

```
#define macro-name      replacement-text         /* Macro Definition */
```

Example 7: Using Functions and Macros

```
/*** Macro Definition ***/
#define mMAX(A,B)       ( (A) > (B) ? (A) : (B) )

/*** Function Definition ***/
char fMAX ( char A, char B)
{
      if ( A > B)             /* Is A Greater than B */
//          CP    R17, R16          ; Compare R16 and R17
//          BRCS  ?fMAX_0           ; Branch if Carry Set
         return A;            /* Yes, Return A */
      else return B;          /* No, Return B */
//          MOV   R16, R17          ; Copy Contents of R17 to R16
// ?fMAX_0:
//          RET                     ; Return from Subroutine
}

__C_task void main (void)
{
      char varA = 15;         /* Initialize varA = 15 */
//          LDI   R16, 15
      char varB = 31;         /* Initialize varB = 31 */
//          LDI   R17, 31
      char result;            /* Allocate Variable result */

      /*** Macro Implementation ***/
      result = mMAX (varA, varB);
//          CP    R17, R16          ; Compare R16 and R17
//          BRCC  ?0001             ; Branch if Carry Cleared
//          MOV   R24, R16          ; Copy Contents of R16 to R24
//          RJMP  ?0002             ; Relative Jump to End
// ?0001:
//          MOV   R24, R17          ; Copy Contents of R17 to R24

      /*** Function Implementation ***/
      result = fMAX (varA, varB);
// ?0002:
//          RCALL fMAX              ; Call fMAX Subroutine
//          MOV   R24, R16          ; Copy Contents of R16 to R24
}
```

The previous code example generates both a macro and function performing the same simple task. As one can see above, the macro implementation in this example is more code efficient than the function implementation. One notices that the function implementation requires significant overhead as `fMAX` needs to called using `RCALL` and then return to the main function after completion using the `RET` instruction. As a general rule, functions, which compile to four lines of assembly code or less can in most cases be more efficiently handled as macros.

Loops

An iteration *loop* simply directs the microcontroller to repeat a certain set of operations or a task until a specific condition is achieved. The C Standards includes three methods for generating iteration loops and each method will be discussed. Example 8 demonstrates the three different loop types operating on the same expression.

A template of a *while* statement is shown below. The first task of the while loop is to evaluate the expression to either true or false (or zero or non-zero). If the expression is true, the statement(s) are executed and returns to the top of the while loop to re-evaluate the expression. The loop continues until the expression result is either zero or false at which point the program jumps to the instruction immediately following the loop.

```
while ( expression )
{
     /* Statement(s) */
}
```

Closely related to while loops are *do…while* loops, as shown below. A do…while loop is slightly different than a while loop, as the statement body is executed at least once. In a while loop, if the expression is false (or zero) initially the statements inside the loop body are never executed. The primary difference between the two is that the test expression resides at the end of the do…while loop instead of the beginning. Once the statement is executed the expression is evaluated, if true it will loop back to the do and execute the expression again, if false the following statement will be executed.

```
do
{
     /* Statement(s) */
} while ( expression );
```

Probably the most common of the iterative loops is the *for* loop. This loop allows the embedded engineer to initialize a variable (expression1), evaluate a condition (expression2), and modify a variable (expression3). Expression1 is executed and evaluated first and is typically an initialization of the variable used in constructing the loop. Expression2 is evaluated next and usually contains the conditional part of the statement, similar to that of the expression in the while and do…while loops. If expression2 is true the statement body is executed, if false execution ceases and the statement following the loop is executed. Finally, expression3 is evaluated and the loop then returns and evaluates expression2 again. One should note that expression1 is only

evaluated initially and expression2 and expression3 are evaluated under each iteration until expression2 is false (or zero).

```
for ( expression1; expression2; expression3 )
{
     /* Statement(s) */
}
```

Example 8: Loops

```
__C_task void main (void)
{
     char counter8 = 0;    /* Initialize counter8 */
//        LDI   R16, 0          ; Initialize R16

     /*** Example: while Loop ***/
     while ( counter8++ < 5 )
// ?0000:
//        MOV   R17, R16        ; Copy Contents of R16 to R17
//        INC   R16             ; Increment R16
//        CPI   R17, 5          ; Compare R16 with Immediate
//        BRNE  ?0000           ; Branch if Not Equal to Zero


     counter8 = 5;
//        LDI   R16, 5          ; Initialize R16

     /*** Example: do...while Loop ***/
     do
     {
     } while ( --counter );
// ?0001
//        DEC   R16             ; Decrement R16
//        BRNE  ?0001           ; Branch if Not Equal to Zero

     /*** Example: for Loop ***/
     for ( counter8 = 0; counter8 < 5; counter++) { }
// ?0002:
//        MOV   R17, R16        ; Copy Contents of R16 to R17
//        INC   R16             ; Increment R16
//        CPI   R17, 5          ; Compare R16 with Immediate
//        BRCC  ?0003           ; Branch if Cary Flag Cleared
//        INC   R16             ; Increment R16
//        RJMP  ?0002           : Relative Jump
// ?0003:
}
```

The loops in Example 8 all perform the same task just using different loops or algorithms. Each loop is executed five times. The most efficient loop is the do…while

loop with pre-decrement, closely followed by the generic while loop with post-increment. As a general rule, do…while loops are the most code efficient, however infinite loops are most efficiently implemented using `for (;;) { }`. Loop counters are generally most efficient when pre-decremented or post-incremented, as branches are dependent upon the status flags after decrement/increment.

Mixing C and Assembly

Most compilers provide the feature of incorporating mixing assembly along with ones high-level C source code. Mixing assembly allows the embedded engineer to code a specific function, sub-routine, interrupt service routine, etc. in assembly while programming the remaining source in a high-level language. This feature is can be beneficial depending on the application, a timing sensitive loop or routine can be easily created mixing assembly and C. Careful planning needs to be taken considering the resources of the microcontroller, such as scratch registers and data memory, when mixing assembly. If one plans on mixing assembly one should read the compiler's manual for proper handling of assembly. Example 9 shows a simple assembly example where the assembly resides in a secondary file included upon compile time.

Example 9: Mixing C and Assembly

```
#include <ioavr.h>

extern void get_port ( void );   /* Prototype for ASM Function */

__C_task void main ( void )
{
      DDRD = 0x00;               /* Initialize PORTD as Output */
      DDRB = 0xFF;               /* Initialize PORTB as Inputs */

      for (;;)                   /* Infinite Loop */
         get_port();             /* Call the ASM Function */
}


NAME get_port
      #include <ioavr.h>         ; #include Must be within the Module
      PUBLIC get_port            ; Declare Symbols to be Exported to C
      RSEG CODE                  ; This Code is Re-locatable, RSEG

get_port:                        ; Label, Start Execution Here
      IN   R16, PIND             ; Read in the PIND Value
      SWAP R16                   ; Swap the Upper and Lower Nibble
      OUT  PORTB, R16            ; Output the Data to PORTD
      RET                        ; Return to the main Function
END
```

Alternatively one can use inline assembly. Inline assembly simply inserts the supplied assembler statement inline. These statements can incorporate both instruction and register mnemonics, constants, and/or variables. One should note however that utilizing

inline assembly severely handicaps the compliers capability of performing optimizaion. If coding in assembly is required, it is recommended to use the approach presented in Example 9 as optimization is sacrificed. The following code snippet demonstrates how one would implement the `get_port` function from Example 9 using inline assembly, it should be noted that inline assembly commands are compiler specific and would need change if one were to port the code to another compiler and/or architecture.

```
asm (“
       get_port:\n
          IN   R16, PIND\n
          SWAP R16\n
          OUT  PORTB, R16\n
          RET\n
    “);
```

Conclusion

One can see that coding embedded source in a high-level language rather than assembly is beneficial. This paper was meant to provide an overview of the advantages of a high-level language while providing some tips on how to implement source code efficiently. More information on the topics discussed in this paper can be found in either application notes available from microcontroller manufacturers and compiler vendors and syntactical information can generally be found from any C programming reference book available online or at your local bookstore. Hopefully, when one writes their next project in a high-level language they remember and implement some of the optimization tips and topics presented in this paper.

References

[1] ATMEL Corporation: Application Note; AVR034: Mixing C and Assembly Code with IAR Embedded Workbench for AVR, ATMEL Document 1234B-AVR, April 2003.

[2] ATMEL Corporation: Application Note; AVR035: Efficient C Coding for AVR, ATMEL Document 1479C-AVR, April 2003.

[3] Brian W. Kernighan and Dennis M. Ritchie: The C Programming Language, Prentice Hall PTR, Englewood Cliffs, New Jersey, 1988.

[4] IAR Systems: Reference Guide; AVR IAR C/EC++ Compiler Reference Guide, IAR Systems Document CAVR-2, March 2002.

[5] Jakob Engblom: Getting the Least Out of Your C Compiler, IAR Systems, Uppsala, Sweden, 2001.

[6] Peter Darnell and Philip E. Margolis: C A Software Engineering Approach, Springer-Verlag, New York, New York, 1996.

[7] Richard Mann: How to Program an 8-bit Microcontroller using C Language, in Proceedings of the Embedded World 2004 Conference, Nurnberg, Germany, February 2004.