Efficient C Coding for FPSLIC™ Using IAR

Features

- **Accessing I/O Memory Locations**
- **Accessing Data in Flash**
- **Efficient Use of Variables and Data Types**
- **Use of Bit-field and Bit-mask**
- **Use of Macros and Functions**
- **Sixteen Ways to Reduce Code Size**
- **Five Ways to Reduce RAM Requirements**
- **Checklist for Debugging Programs**

Introduction

The C High-level Language (HLL) has become increasingly popular for programming microcontrollers. The advantages of using C compared to assembler are numerous: reduced development time, easier maintainability and portability and easier to reuse code. The penalty is larger code size and, as a result of that, often reduced speed. To reduce these penalties, the FPSLIC architecture is tuned to efficiently decode and execute instructions that are typically generated by C compilers.

The C compiler development was done by IAR Systems before the FPSLIC architecture and instruction set specifications were completed. The result of the cooperation between the compiler development team and the FPSLIC development team is a microcontroller for which highly efficient, high-performance code is generated.

This application note describes how to utilize the advantages of the FPSLIC architecture and the development tools to achieve more efficient C code than for any other microcontroller.

Architecture Tuned for C Code

The thirty-two working registers are one of the keys to efficient C coding. These registers have the same function as the traditional accumulator, except that there are thirty-two of them. In one clock cycle, AVR can feed two arbitrary registers from the register file to the ALU, perform an operation and write back the result to the register file.

When data are stored in the thirty-two working registers, there is no need to move the data to and from memory between each arithmetic instruction. Some of the registers can be combined to 16-bit pointers that efficiently access data in the data and program memories. For large memory sizes, the memory pointers can be combined with a third 8-bit register to form 24-bit pointers that can access 16M bytes of data, with no paging!

Addressing Modes

The FPSLIC microcontroller architecture has four memory pointers that are used to access data and program memory. The stack pointer (SP) is dedicated for storing the return address after return from a function. The C compiler allocates one pointer as parameter stack. The two remaining pointers are general-purpose pointers used by the C compiler to load and store data. The example below shows how efficiently the pointers are used for typical pointer operations in C.

```
char *pointer1 = &table[0];
char *pointer2 = &table[49];


*pointer1++ = *--pointer2;
```

FPSLIC™

10K - 40K Gates of AT40K FPGA with 8-bit AVR® Microcontroller and 36K Bytes of SRAM

Application Note

Rev. 1975A–11/00

This generates the following assembly code:

```
LD  R16,-Z; Pre-decrement Z pointer and load data
ST  X+,R16; Store data and post increment
```

The four pointer addressing modes and examples are shown below. All pointer operations are single-word instructions that execute in two clock cycles.

1. Indirect addressing. For addressing of arrays and pointer variables:

   ```
   *pointer = 0x00;
   ```

2. Indirect addressing with displacement. Allows accesses to all elements in a structure by pointing to the first element and add displacement without having to change the pointer value. Also used for accessing variables on the software stack and array accesses.
3. Indirect addressing with post-increment. For efficient addressing of arrays and pointer variables with increment after access:

   ```
   *pointer++ = 0xFF;
   ```

4. Indirect addressing with pre-decrement. For efficient addressing of arrays and pointer variables with decrement before access:

   ```
   *--pointer = 0xFF
   ```

The pointers are also used to access the Flash program memory. In addition to indirect addressing with pointers, the data memory can also be accessed by direct addressing. This gives access to the entire data memory in a two-word instruction.

Support for 16/32-bit Variables

The FPSLIC instruction set includes special instructions for handling 16-bit numbers. This includes Add/Subtract Immediate Values to Word (ADIW, SBIW). Arithmetic operations and comparison of 16-bit numbers are completed with two instructions in two clock cycles. 32-bit arithmetic operations and comparison are ready in four instructions and four cycles. This is more efficient than most 16-bit processors!

C Code for AVR

Initializing the Stack Pointer

After power-up or RESET, the stack pointer needs to be set up before any function is called. The linker command file determines the placement and size of the stack pointer. The configuration of memory sizes and stack pointer setup is explained in Atmel's application note "AVR032: Linker Command Files for the IAR ICCA90 Compiler".

Accessing I/O Memory Locations

The AVR I/O memory is easily accessed in C. All registers in the I/O memory are declared in a header file usually named "ioxxxx.h", where xxxx is the FPSLIC part number. The code below shows examples of accessing I/O location. The assembly code generated for each line is shown below each C code line.

```
#include <ioat94k.h>                          /* Include header file with symbolic names */

void C_task main(void)
{
  char temp;                                  /* Declare a temporary variable*/

/*To read and write to an I/O register*/
  temp = PIND;                                /* Read PIND into a variable*/
//      IN      R16,LOW(16)                   ; Read I/O memory

  TCCR0 = 0x4F;                               /* Write a value to an I/O location*/
//      LDI     R17,79                        ; Load value
//      OUT     LOW(51),R17                   ; Write I/O memory

/*Set and clear a single bit */
  PORTD |= (1<<PIND2);                        /* PIND2 is pin number(0..7)in port */
//      SBI     LOW(24),LOW(2)                ; Set bit in I/O

  ADCSR &= ~(1<<ADEN);                        /* Clear ADEN bit in ADCSR register */
//      CBI     LOW(6),LOW(7)                 ; Clear bit in I/O
/* Set and clear a bitmask*/
```

```
  DDRD |= 0x0C;                                  /* Set bit 2 and 3 in DDRD register*/
//      IN      R17,LOW(17)                     ; Read I/O memory
//      ORI     R17,LOW(12)                     ; Modify
//      OUT     LOW(17),R17                     ; Write I/O memory

  ACSR &= ~(0x0C);                               /* Clear bit 2 and 3 in ACSR register*/
//      IN      R17,LOW(8)                      ; Read I/O memory
//      ANDI    R17,LOW(243)                    ; Modify
//      OUT     LOW(8),R17                      ; Write I/O memory

/* Test if a single bit is set or cleared */
  if(USR & (1<<TXC))                             /* Check if UART Tx flag is set*/
  {
    PORTE |= (1<<PE0);

//      SBIC    LOW(11),LOW(6)                  ; Test direct on I/O
//      SBI     LOW(24),LOW(0)

  while(!(SPSR & (1<<WCOL)));                    /* Wait for WCOL flag to be set */
//      ?0003:SBIS    LOW(14),LOW(6)            ; Test direct on I/O
//      RJMP    ?0003

/* Test if an I/O register equals a bitmask */
  if(UDR & 0xF3)                                 /* Check if UDR register "and" 0xF3 is non-zero */
  {
  }
//      IN      R16,LOW(12)                     ; Read I/O memory
//      ANDI    R16,LOW(243)                    ; "And" value
//      BREQ    ?0008                           ; Branch if equal

//?0008:
}
/* Set and clear bits in I/O registers can also be declared as macros */

#define SETBIT(ADDRESS,BIT) (ADDRESS |= (1<<BIT))
#define CLEARBIT(ADDRESS,BIT) (ADDRESS &= ~(1<<BIT))

/* Macro for testing of a single bit in an I/O location*/
#define CHECKBIT(ADDRESS,BIT) (ADDRESS & (1<<BIT))

/* Example of usage*/
if(CHECKBIT(PORTD,PIND1))                        /* Test if PIN 1 is set*/
{
       CLEARBIT(PORTD,PIND1);                    /* Clear PIN 1 on PORTD*/
}
if(!(CHECKBIT(PORTD,PIND1)))                     /* Test if PIN 1 is cleared*/
{
    SETBIT(PORTD,PIND1);                         /* Set PIN 1 on PORTD*/
}
```

Variables and Data Types

Data Types

As the FPSLIC is an 8-bit microcontroller, use of 16- and 32-bit variables should be limited to where it is absolutely necessary. The following example shows the code size for a loop counter for an 8-bit and 16-bit local variable.

8-bit Counter

```
unsigned char count8 = 5;                    /* Declare a varible, assign a value    */
//    LDI     R16,5                          ;Init variable

do                                           /* Start a loop                         */
{
}while(--count8);                            /* Decrement loop counter and check for zero */
//      ?0004:DEC     R16                    ; Decrement
//      BRNE   ?0004                         ; Branch if not equal
```

16-bit Counter

```
unsigned int count16 = 6;                    /* Declare a variable, assign a value   */
//      LDI     R24,LOW(6)                   ;Init variable, low byte
//      LDI     R25,0                        ;Init variable, high byte

do                                           /* Start a loop          */
{
}while(--count16);                           /* Decrement loop counter and check for zero  */
//      ?0004:SBIW    R24,LWRD(1)            ; Subtract 16-bit value
//      BRNE    ?0004                        ; Branch if not equal
```

Table 1. Variable and Code Size

Variable	Code Size (Bytes)
8-bit	6
16-bit	8

Note: Always use the smallest applicable variable type. This is especially important for global variables.

Efficient Use of Variables

A C program is divided into many functions that execute small or big tasks. The functions receive data through parameters and may also return data. Variables used inside the function are called local variables. Variables declared outside a function are called global variables. Variables that are local, but must be preserved between each time the function is used, should be declared as static local variables.

Global variables that are declared outside a function are assigned to an SRAM memory location. The SRAM location is reserved for the global variable and cannot be used for other purposes – this is considered to be a waste of valuable SRAM space. Too many global variables make the code less readable and hard to modify.

Local variables are preferably assigned to a register when they are declared. The local variable is kept in the same register until the end of the function or until it is not referenced further. Global variables must be loaded from the SRAM into the working registers before they are accessed.

The following example illustrates the difference in code size and execution speed for local variables compared to global variables.

```
char global;                                    /* This is a global variable */

void C_task 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
//      SUBI    R16,LOW(45)                     ; Perform subtraction
//      STS     LWRD(global),R16                ; Store data back in SRAM

  local -= 34;                                  /* Subtraction with local variable*/
//      SUBI    R16,LOW(34)                     ; Perform subtraction directly on local variable in
                                                register R16

}
```

Note that the LDS and STS (Load and Store direct from/to SRAM) are used to access the variables in SRAM. These are two-word instructions that execute in two cycles.

Table 2. Code Size and Execution Time for Variables

Variable	Code Size (Bytes)	Execution Time (Cycles)
Global	10	5
Local	2	1

A local static variable is loaded into a working register at the start of the function and stored back to its SRAM location at the end of the function. Static variables will therefore give more efficient code than global variables if the variable is accessed more than once inside the function.

To limit the use of global variables, functions can be called with parameters and return a value commonly used in C. Up to two parameters of simple data types (char, int, float, double) are passed between functions in the registers R16 - R23. More than two parameters and complex data types (arrays, structs) are either placed on the software stack or passed between functions as pointers to SRAM locations.

When global variables are required, they should be collected in structures whenever appropriate. This makes it possible for the C compiler to address them indirectly. The following example shows the code generation for global variable versus global structures.

```
typedef struct
{
        char sec;
}t;
t global                                        /* Declare a global structure*/

char min;

void C_task main(void)
{
  t *time = &global;
//      LDI     R30,LOW(global)                 ; Init Z pointer
//      LDI     R31,(global >> 8)               ; Init Z high byte

  if (++time->sec == 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
  }
  if ( ++min ==  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    ?0005                           ; Branch if not equal
  }
}
```

When accessing the global variables as structures, the compiler is using the Z-pointer and the LDD and STD (Load/Store with displacement) instructions to access the data. When the global variables are accessed without structures, the compiler uses LDS and STS (Load/Store direct to SRAM). The difference in code size is shown in Table 3.

Table 3. Code Size for Global Variables

Variable	Code Size (Bytes)
Structure	10
Non-structure	14

This does not include initialization of the Z-pointer (4 bytes) for the global structure. To access one byte, the code size will be the same, but if the structure consists of two bytes or more it will be more efficient to access the global variables in a structure.

Unused locations in the I/O memory can be utilized for storing global variables when certain peripherals are not used. For example, if the UART1 is not used, the UART1 Baud Rate Register (UBRR1) is available to store global variables.

The I/O memory is accessed very efficiently and locations below 0x1F in the I/O memory are especially suited since they are bit-accessible.

Bit-field versus Bit-mask

To save valuable bytes of data storage it may be necessary to save several single-bit flags into one byte. A common use of this is bit flags that are packed in a status byte. This can either be defined as bit-mask or bit-field. Below is an example of the use of bit-mask and bit-field to declare a status byte.

```
/* Use of bit-mask for status bits*/

/* Define bit macros, note that they are similar to the I/O macros*/

#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*/

/* Define Status bit mask constants */

#define RETRANS  0x01                           /* bit 0 : Retransmit Flag*/
#define WRITEFLAG 0x02                          /* bit 1 : Flag set when write is due*/
#define EMPTY     0x04                          /* bit 2 : Empty buffer flag*/
#define FULL      0x08                          /* bit 3 : Full buffer flag*/

void C_task main(void)
{
```

```
  char status;                                  /* Declare a status byte*/

  CLEARBIT(status,RETRANS);                     /* Clear RETRANS and WRITEFLAG*/
  CLEARBIT(status,WRITEFLAG);

/*Check if RETRANS flag is cleared */
  if (!(CHECKBIT(status, RETRANS)))
  {
    SETBIT(status,WRITEFLAG);
}
}
```

Bit-masks are handled very efficiently by the C compiler if the status variable is declared as local variable within the function it is used. Alternatively, use unused I/O locations with bit-mask.

The same function with bit-fields:

```
/* Use of bit-fields for status bits*/

void C_task main(void)
{
        struct {
        char RETRANS: 1 ;                       /* bit 0 : Retransmit Flag*/
        char WRITEFLAG   : 1 ;                  /* bit 1 : Flag set when write is due */
        char EMPTY      : 1 ;                   /* bit 2 : Empty buffer flag*/
        char FULL      : 1 ;                    /* bit 3 : Full buffer flag*/
        } status;                               /* Declare a status byte*/


        status.RETRANS = 0;                     /* Clear RETRANS and WRITEFLAG*/
        status.WRITEFLAG = 0;

        if (!(status.RETRANS))                  /* Check if RETRANS flag is cleared*/
        {
          status.WRITEFLAG = 1;
        }
}
```

Bit-fields are not stored locally in the register file within the function, but popped and pushed on the code stack each time it is accessed. Therefore, the code generated with bit-masks is more efficient and faster than using bit-fields. The ANSI standard does not define how bit-fields are packed into the byte, i.e., a bit-field placed in the MSB (Most Significant Bit) with one compiler can be placed in the LSB (Least Significant Bit) in another compiler. With bit-mask, the user has complete control of the bit placement inside the variables.

Accessing Flash Memory

A common way to define a constant is:

```
const char max = 127;
```

This constant is copied from Flash memory to SRAM at start-up and remains in the SRAM for the rest of the program execution. This is considered to be a waste of SRAM. To save SRAM space, the constant can be saved in Flash and loaded when it is needed.

```
flash char max = 127;
flash char string[] = "This string is stored in flash";

void main(void)
{
  char flash *flashpointer;                    ; Declare flash pointer

  flashpointer = &string[0];                   ; Assign pointer to flash location
  UDR1 = *flashpointer;                        ; Read data from flash and write to UART1
}
```

When strings are stored in Flash, like in the latter example, they can be accessed directly or through pointers to the Flash program memory. For the IAR C compiler, special library routines exist for string handling. See the *IAR Compiler User's Manual* for details.

Control Flow

The Main Function

The main function usually contains the main loop of the program. In most cases, no functions are calling the main function, and there is no need to preserve any registers when entering it. The main function can therefore be declared as C_task. This saves stack space and code size.

```
void C_task main(void)                         /* Declare main() as C_task*/
{

}
```

Loops

Eternal loops are most efficiently constructed using for(; ;) { }:

```
for( ; ;)
{
      /* This is an eternal loop*/
}
//    ?0001:RJMP ?0001                         ; Jump to label
```

do{ }while(expression) loops generally generates more efficient code than while{ } and *for{expr1; expr2; expr3)*. The following example shows the code generated for a *do{ } while* loop:

```
char counter = 100;                            /* Declare loop counter variable*/
//      LDI  R16,100                           ; Init variable
do
{
} while(--counter);                            /* Decrement counter and test for zero*/
      ?0004:DEC   R16                          ; Decrement
//      BRNE  ?0004                            ; Branch if not equal
```

Pre-decrement variables such as loop counter usually give the most efficient code. Pre-decrement and post-increment is more efficient because branches are depending on the flags after decrement.

Macros versus Functions

Functions that assemble into 3 - 4 lines or less of assembly code can in some cases be handled more efficiently as macros. When using macros, the macro name will be replaced by the actual code inside the macro at compile time. For very small functions, the compiler generates less code and gives higher speed to use macros than it does to call a function.

The example below shows how a task can be executed in a function and as a macro.

```
/* Main function to call the task*/
void C_task main(void)
{
  UDR1 = read_and_convert();                    /* Read value and write to UART1*/
}

/* Function to read pin value and convert it to ASCII*/
char read_and_convert(void)
{
  return (PINE + 0x48);                         /* Return the value as ASCII character   */
}

/* A macro to do the same task*/
#define read_and_convert (PINE + 0x48)
```

The code with function assembles into the following code:

```
main:
//          RCALL   read_and_convert            ; Call function
//          OUT     LOW(12),R16                 ; Write to I/O memory

read_and_convert:
//          IN      R16,LOW(22)                 ; Read I/O memory
//          SUBI    R16,LOW(184)                ; Add 48 to value
//          RET                                 ; Return
```

The code with macro assembles into this code:

```
main:
//      IN      R16,LOW(22)                     ; Read I/O memory
//      SUBI    R16,LOW(184)                    ; Add 48 to value
//      OUT     LOW(12),R16                     ; Write I/O memory
```

Table 4. Code Size and Execution Time for Macros and Functions

Variable	Code Size (Bytes)	Execution Time (Cycles)
Function	10	10
Macro	6	3

Sixteen Hints to Reduce Code Size

1. Compile with full-size optimization.
2. Use local variables whenever possible.
3. Use the smallest applicable data type. Use unsigned, if applicable.
4. If a non-local variable is only referenced within one function, it should be declared static.
5. Collect non-local data in structures whenever natural. This increases the possibility of indirect addressing without pointer reload.
6. Use for(; ;) { } for eternal loops.
7. Use do { } while(expression) if applicable.
8. Use descending loop counters and pre-decrement if applicable.
9. Access I/O memory directly (i.e., do not use pointers).
10. Use bit-masks on unsigned chars or unsigned ints instead of bit-fields.
11. Declare main as C_task if not called from anywhere in the program.
12. Use macros instead of functions for tasks that generate less than 2 - 3 lines of assembly code.
13. Reduce the size of the interrupt vector segment (INTVEC) to what is actually needed by the application. Alternatively, concatenate all the CODE segments into one declaration and it will be done automatically.
14. Code reuse is intra-modular. Collect several functions in one module (i.e., in one file) to increase code reuse factor.
15. In some cases, full-speed optimization results in lower code size than full-size optimization. Compile on a module-by-module basis to investigate what gives the best result.
16. Optimize C_startup to not initialize unused segments (i.e., IDATA0 or IDATA1 if all variables are *tiny* or *small*).

Five Hints to Reduce RAM Requirements

1. All constants and literals should be placed in Flash by using the Flash keyword.
2. Avoid using global variables if the variables are local in nature. This also saves code space. Local variables are allocated from the stack dynamically and are removed when the function goes out of scope.
3. If using large functions with variables with a limited lifetime within the function, the use of subscopes can be beneficial.
4. Get good estimates of the sizes of the software stack and return stack (linker file).
5. Do not waste space for the IDATA0 and UDATA0 segments unless you are using tiny variables (linker file).

Checklist for Debugging Programs

1. Ensure that the CSTACK segment is sufficiently large.
2. Ensure that the RSTACK segment is sufficiently large.
3. If a regular function and an interrupt routine are communicating through a global variable, make sure this variable is declared volatile to ensure that it is reread from RAM each time it is checked.

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL (408) 441-0311
FAX (408) 487-2600

Europe
Atmel SarL
Route des Arsenaux 41
Casa Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Atmel Asia, Ltd.
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
Atmel Japan K.K.
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Atmel Colorado Springs
1150 E. Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL (719) 576-3300
FAX (719) 540-1759

Atmel Rousset
Zone Industrielle
13106 Rousset Cedex
France
TEL (33) 4-4253-6000
FAX (33) 4-4253-6001

Atmel Smart Card ICs
Scottish Enterprise Technology Park
East Kilbride, Scotland G75 0QR
TEL (44) 1355-803-000
FAX (44) 1355-242-743

Atmel Grenoble
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex
France
TEL (33) 4-7658-3000
FAX (33) 4-7658-3480

Atmel FPSLIC Hotline
1-(408) 436-4119

Atmel FPSLIC e-mail
fpslic@atmel.com

FAQ
Available from Website

Fax-on-Demand
North America:
1-(800) 292-8635
International:
1-(408) 441-0732

e-mail
literature@atmel.com

Web Site
http://www.atmel.com

BBS
1-(408) 436-4309

1975A–11/00/xM