Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers

Features

- **Atmel® AVR® core and Atmel AVR GCC introduction**
- **Tips and tricks to reduce code size**
- **Tips and tricks to reduce execution time**
- **Examples application**

1 Introduction

AVR core is an advanced RISC architecture tuned for C code. It ensures the development of good products with more features at less cost.

When talking about optimization, we usually refer to two aspects: code size and code speed. Nowadays, C compilers have different optimization options to help developers get an efficient code on either size or speed.

However, good C coding gives more opportunities for compilers to optimize the code as desired. And in some cases, optimizing for one of the two aspects affects or even causes degradation in the other, so a developer has to balance the two according to their specific needs. An understanding of some tips and tricks about C coding for an 8-bit AVR helps the developers to know where to focus in improving code efficiency.

In this application note, the tips are based on avr-gcc (C compiler). However these tips could be implemented in other compilers or with similar compiler options, and vice versa.

8-bit Atmel Microcontrollers

Application Note

Rev. 8453A-AVR-11/11

2 Knowing Atmel AVR core and Atmel AVR GCC

Before optimizing embedded systems software, it is necessary to have a good understanding of how the AVR core is structured and what strategies the AVR GCC uses to generate efficient code for this processor. Here we have a short introduction of the features of AVR core and AVR GCC.

2.1 Atmel AVR 8-bit architecture

AVR uses Harvard architecture – with separate memories and buses for program and data. It has a fast-access register file of 32 × 8 general purpose working registers with a single clock cycle access time. The 32 working registers is one of the keys to efficient C coding. These registers have the same function as the traditional accumulator, except that there are 32 of them. The AVR arithmetic and logical instructions work on these registers, hence they take up less instruction space. 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.

Instructions in the program memory are executed with a single level pipelining. While one instruction is being executed, the next instruction is pre-fetched from the program memory. This concept enables instructions to be executed in every clock cycle. Most AVR instructions have a single 16-bit word format. Every program memory address contains a 16- or 32-bit instruction.

Please refer to “AVR CPU Core” section in device datasheet for more details.

2.2 AVR GCC

GCC stands for GNU Compiler Collection. When GCC is used for the AVR target, it is commonly known as AVR GCC. The actual program “gcc” is prefixed with "avr-", namely, “avr-gcc”.

AVR GCC provides several optimization levels. They are -O0, -O1, -O2, -O3 and -Os. In each level, there are different optimization options enabled, except for -O0 which means no optimization. Besides the options enabled in optimization levels, you can also enable separate optimization options to get a specific optimization.

Please refer to the GNU Compiler Collection manual as below for a complete list of optimization options and levels.

http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options

Besides “avr-gcc”, it takes many other tools working together to produce the final executable application for the AVR microcontroller. The group of tools is called a toolchain. In this AVR toolchain, avr-libc serves as an important C Library which provides many of the same functions found in a regular Standard C Library and many additional library functions that is specific to an AVR.

The AVR Libc package provides a subset of the standard C library for Atmel AVR 8-bit RISC microcontrollers. In addition, the library provides the basic startup code needed by most applications.

Please check the link below for the manual of avr-libc,

http://www.nongnu.org/avr-libc/user-manual/

2.3 Development platform

The example codes and testing results in this document are based on the following platform and device,

1. Integrated Development Environment (IDE):
   Atmel AVR Studio® 5 (Version: 5.0.1119).
2. AVR GCC 8-bit Toolchain Version:
   AVR_8_bit_GNU_Toolchain_3.2.1_292 (gcc version 4.5.1).
3. Target Device:
   Atmel ATmega88PA.

3 Tips and tricks to reduce code size

In this section, we list some tips about how to reduce code size. For each tip description and sample code are given.

3.1 Tip #1 – Data types and sizes

Use the smallest applicable data type as possible. Evaluate your code and in particular the data types. Reading an 8-bit (byte) value from a register only requires a byte-sized variable and not a double-byte variable, thus saving code-space.

The size of data types on 8-bit AVR can be found in the <stdint.h> header file and is summarized in Table 3-1.

Table 3-1. Data types on 8-bit AVR in <stdint.h>.

Data type		Size
signed char / unsigned char	int8_t / uint8_t	8-bit
signed int / unsigned int	int16_t / uint16_t	16-bit
signed long / unsigned long	int32_t / uint32_t	32-bit
signed long long / unsigned long long	int64_t / uint64_t	64-bit

Be aware that certain compiler-switches can change this (avr-gcc -mint8 turns integer data type to be 8-bit integer).

The two example codes in Table 3-2 show the effect of different data types and sizes. The output from the avr-size utility shows the code space we used when this application is built with -Os (optimize for size).

Table 3-2. Example of different data types and sizes.

	Unsigned int (16-bit)	**Unsigned char (8-bit)**

C source code	`#include <avr/io.h>`
	
	`unsigned int readADC() {`
	`  return ADCH;`
	`};`
	
	`int main(void)`
	`{`
	`  unsigned int mAdc = readADC();`
	`}`
		`#include <avr/io.h>`
		
		`unsigned char readADC() {`
		`  return ADCH;`
		`};`
		
		`int main(void)`
		`{`
		`  unsigned char mAdc = readADC();`
		`}`

AVR Memory Usage	Program: 92 bytes (1.1% full)	Program: 90 bytes (1.1% full)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

In the left example, we use the int (2-byte) data type as return value from the readADC() function and in the temporary variable used to store the return value from the readADC() function.

In the right example we are using char (1-byte) instead. The readout from the ADCH register is only eight bits, and this means that a char is sufficient. Two bytes are saved due to the return value of the function readADC() and the temporary variable in main being changed from int (2-byte) to char (1-byte).

NOTE

There is a startup code before running from main(). That's why a simple C code takes up about 90 bytes.

3.2 Tip #2 – Global variables and local values

In most cases, the use of global variables is not recommended. Use local variables whenever possible. If a variable is used only in a function, then it should be declared inside the function as a local variable.

In theory, the choice of whether to declare a variable as a global or local variable should be decided by how it is used.

If a global variable is declared, a unique address in the SRAM will be assigned to this variable at program link time. Also accessing to a global variable will typically need extra bytes (usually two bytes for a 16 bits long address) to get its address.

Local variables are preferably assigned to a register or allocated to stack if supported when they are declared. As the function becomes active, the function's local variables become active as well. Once the function exits, the function's local variables can be removed.

In Table 3-3 there are two examples showing the effect of global variables and local variables.

Table 3-3. Example of global variables and local variables.

	Global variables	**Local variables**

C source code	`#include <avr/io.h>`
	
	`uint8_t global_1;`
	
	`int main(void)`
	`{`
	`  global_1 = 0xAA;`
	`  PORTB = global_1;`
	`}`
		`#include <avr/io.h>`
		
		`int main(void)`
		`{`
		`  uint8_t local_1;`
		
		`  local_1 = 0xAA;`
		`  PORTB = local_1;`
		`}`

AVR Memory Usage	Program: 104 bytes (1.3% full) (.text + .data + .bootloader) Data: 1 byte (0.1% full) (.data + .bss + .noinit)	Program: 84 bytes (1.0% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

In the left example, we have declared a byte-sized global variable. The output from the avr-size utility shows that we use 104 bytes of code space and one byte of data space with optimization level -Os (optimize for size).

In the right example, after we declared the variable inside main() function as local variable, the code space is reduced to 84 bytes and no SRAM is used.

3.3 Tip #3 – Loop index

Loops are widely used in 8-bit AVR code. There are “while ( ) { }” loop, “for ( )” loop and “do { } while ( )” loop. If the -Os optimization option is enabled; the compiler will optimize the loops automatically to have the same code size.

However we can still reduce the code size slightly. If we use a “do { } while ( )” loop, an increment or a decrement loop index generates different code size. Usually we write our loops counting from zero to the maximum value (increment), but it is more efficient to count the loop from the maximum value to zero (decrement).

That is because in an increment loop, a comparison instruction is needed to compare the loop index with the maximum value in every loop to check if the loop index reaches the maximum value.

When we use a decrement loop, this comparison is not needed any more because the decremented result of the loop index will set the Z (zero) flag in SREG if it reaches zero.

In Table 3-4 there are two examples showing the code generated by “do { } while ( )” loop with increment and decrement loop indices. The optimization level -Os (optimize for size) is used here.

Table 3-4. Example of do { } while ( ) loops with increment and decrement loop index.

	do{ }while( ) with increment loop index	**do{ }while( ) with decrement loop index**

C source code	`#include <avr/io.h>`
	
	`int main(void)`
	`{`
	`  uint8_t local_1 = 0;`
	
	`  do {`
	`  PORTB ^= 0x01;`
	`  local_1++;`
	`  } while (local_1<100);`
	`}`
		`#include <avr/io.h>`
		
		`int main(void)`
		`{`
		`  uint8_t local_1 = 100;`
		
		`  do {`
		`  PORTB ^= 0x01;`
		`  local_1--;`
		`  } while (local_1);`
		`}`

AVR Memory Usage	Program: 96 bytes (1.2% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)	Program: 94 bytes (1.1% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

To have a clear comparison in C code lines, this example is written like “do {count-- ;} while (count);” and not like “do {} while (--count);” usually used in C books. The two styles generate the same code.

3.4 Tip #4 – Loop jamming

Loop jamming here refers to integrating the statements and operations from different loops to fewer loops or to one loop, thus reduce the number of loops in the code.

In some cases, several loops are implemented one by one. And this may lead to a long list of iterations. In this case, loop jamming may help to increase the code efficiency by actually having the loops combined into one.

Loop jamming reduces code size and makes code run faster as well by eliminating the loop iteration overhead. From the example in Table 3-5, we could see how loop jamming works.

Table 3-5. Example of loop jamming.

	Separate loops	**Loop jamming**

C source code	`#include <avr/io.h>`
	
	`int main(void)`
	`{`
	`  uint8_t i, total = 0;`
	`  uint8_t tmp[10] = {0};`
	
	`  for (i=0; i<10; i++) {`
	`  tmp [i] = ADCH;`
	`  }`
	`  for (i=0; i<10; i++) {`
	`  total += tmp[i];`
	`  }`
	`  UDR0 = total;`
	`}`
		`#include <avr/io.h>`
		
		`int main(void)`
		`{`
		`  uint8_t i, total = 0;`
		`  uint8_t tmp[10] = {0};`
		
		`  for (i=0; i<10; i++) {`
		`  tmp [i] = ADCH;`
		`  total += tmp[i];`
		`  }`
		`  UDR0 = total;`
		`}`

AVR Memory Usage	Program: 164 bytes (2.0% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)	Program: 98 bytes (1.2% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

3.5 Tip #5 – Constants in program space

Many applications run out of SRAM, in which to store data, before they run out of Flash. Constant global variables, tables or arrays which never change, should usually be allocated to a read-only section (Flash or EEPROM on 8-bit AVR) and. This way we can save precious SRAM space.

In this example we don't use C keyword "const". Declaring an object "const" announces that its value will not be changed. "const" is used to tell the compiler that the data is to be "read-only" and increases opportunities for optimization. It does not identify where the data should be stored.

To allocate data into program space (read-only) and receive them from program space, AVR-Libc provides a simple macro "PROGMEM" and a macro "pgm_read_byte". The PROGMEM macro and pgm_read_byte function are defined in the <avr/pgmspace.h> system header file.

The following example in Table 3-6 show how we save SRAM by moving the global string into program space.

Table 3-6. Example of constants in program space.

	Constants in data space	**Constants in program space**

C source code	`#include <avr/io.h>`
	
	`uint8_t string[12] = {"hello world!"};`
	
	`int main(void)`
	`{`
	`  UDR0 = string[10];`
	`}`
		`#include <avr/io.h>`
		`#include <avr/pgmspace.h>`
		
		`uint8_t string[12] PROGMEM = {"hello world!"};`
		
		`int main(void)`
		`{`
		`  UDR0 = pgm_read_byte(&string[10]);`
		`}`

AVR Memory Usage	Program: 122 bytes (1.5% full) (.text + .data + .bootloader) Data: 12 bytes (1.2% full) (.data + .bss + .noinit)	Program: 102 bytes (1.2% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

After we allocate the constants into program space, we see that the program space and data space are both reduced. However, there is a slight overhead when reading back the data, because the function execution will be slower than reading data from SRAM directly.

If the data stored in flash are used multiple times in the code, we get a lower size by using a temporary variable instead of using the "pgm_read_byte" macro directly several times.

There are more macros and functions in the <avr/pgmspace.h> system header file for storing and retrieving different types of data to/from program space. Please check avr-libc user manual for more details.

3.6 Tip #6 – Access types: Static

For global data, use the static keyword whenever possible. If global variables are declared with keyword static, they can be accessed only in the file in which they are defined. It prevents an unplanned use of the variable (as an external variable) by the code in other files.

On the other hand, local variables inside a function should be avoided being declared as static. A “static” local variable’s value needs to be preserved between calls to the function and the variable persists throughout the whole program. Thus it requires permanent data space (SRAM) storage and extra codes to access it. It is similar to a global variable except its scope is in the function where it’s defined.

A static function is easier to optimize, because its name is invisible outside of the file in which it is declared and it will not be called from any other files.

If a static function is called only once in the file with optimization (-O1, -O2, -O3 and -Os) enabled, the function will be optimized automatically by the compiler as an inline function and no assembler code is outputted for this function. Please check the example in Table 3-7 for the effect.

Table 3-7. Example of access types: static function.

	Global function (called once)	**Static function (called once)**

C source code	`#include <avr/io.h>`
	
	`uint8_t string[12] = {"hello world!"};`
	
	`void USART_TX(uint8_t data);`
	
	`int main(void)`
	`{`
	`  uint8_t i = 0;`
	`  while (i<12) {`
	`  USART_TX(string[i++]);`
	`  }`
	`}`
	
	`void USART_TX(uint8_t data)`
	`{`
	`  while(!(UCSR0A&(1<<UDRE0)));`
	`  UDR0 = data;`
	`}`
		`#include <avr/io.h>`
		
		`uint8_t string[12] = {"hello world!"};`
		
		`static void USART_TX(uint8_t data);`
		
		`int main(void)`
		`{`
		`  uint8_t i = 0;`
		`  while (i<12) {`
		`  USART_TX(string[i++]);`
		`  }`
		`}`
		
		`void USART_TX(uint8_t data)`
		`{`
		`  while(!(UCSR0A&(1<<UDRE0)));`
		`  UDR0 = data;`
		`}`

AVR Memory Usage	Program: 152 bytes (1.9% full) (.text + .data + .bootloader) Data: 12 bytes (1.2% full) (.data + .bss + .noinit)	Program: 140 bytes (1.7% full) (.text + .data + .bootloader) Data: 12 bytes (1.2% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

NOTE If the function is called multiple times, it will not be optimized to an inline function, because this will generate more code than direct function calls.

3.7 Tip #7 – Low level assembly instructions

Well coded assembly instructions are always the best optimized code. One drawback of assembly code is the non-portable syntax, so it's not recommended for programmers in most cases.

However, using assembly macros reduces the pain often associated with assembly code, and it improves the readability and portability. Use macros instead of functions for tasks that generates less than 2-3 lines assembly code. The example in Table 3-8 shows the code usage of assembly macro compared with using a function.

Table 3-8. Example of low level assembly instructions.

	Function	**Assembly macro**

C source code	`#include <avr/io.h>`
	
	`void enable_usart_rx(void)`
	`{`
	`  UCSR0B |= 0x80;`
	`};`
	
	`int main(void)`
	`{`
	`  enable_usart_rx();`
	`  while (1){`
	`  }`
	`}`
		`#include <avr/io.h>`
		
		`#define enable_usart_rx()  \`
		`__asm__ __volatile__ (  \`
		`  "lds  r24,0x00C1" "\n\t"  \`
		`  "ori  r24, 0x80" "\n\t"  \`
		`  "sts  0x00C1, r24"  \`
		`  ::)`
		
		`int main(void)`
		`{`
		`  enable_usart_rx();`
		`  while (1){`
		`  }`
		`}`

AVR Memory Usage	Program: 90 bytes (1.1% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)	Program: 86 bytes (1.0% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Compiler optimization level	-Os (optimize for size)	-Os (optimize for size)

For more details about using assembly language with C in 8-bit AVR, please refer to "Inline Assembler Cookbook" section in avr-libc user manual.

4 Tips and tricks to reduce execution time

In this section, we list some tips about how to reduce execution time. For each tip, some description and sample code are given.

4.1 Tip #8 – Data types and sizes

In addition to reducing code size, selecting a proper data type and size will reduce execution time as well. For 8-bit AVR, accessing 8-bit (Byte) value is always the most efficient way.

Please check the example in Table 4-1 for the difference of 8-bit and 16-bit variables.

Table 4-1. Example of data types and sizes.

	16-bit variable	**8-bit variable**

C source code	`#include <avr/io.h>`
	
	`int main(void)`
	`{`
	`  uint16_t local_1 = 10;`
	
	`  do {`
	`  PORTB ^= 0x80;`
	`  } while (--local_1);`
	`}`
		`#include <avr/io.h>`
		
		`int main(void)`
		`{`
		`  uint8_t local_1 = 10;`
		
		`  do {`
		`  PORTB ^= 0x80;`
		`  } while (--local_1);`
		`}`

AVR Memory Usage	Program: 94 bytes (1.1% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)	Program: 92 bytes (1.1% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Cycle counter	90	79
Compiler optimization level	-O2	-O2

NOTE The loop will be unrolled by compiler automatically with –O3 option. Then the loop will be expanded into repeating operations indicated by the loop index, so for this example there is no difference with –O3 option enabled.

4.2 Tip #9 – Conditional statement

Usually pre-decrement and post-decrement (or pre-increments and post-increments) in normal code lines make no difference. For example, “i--;” and “--i;” simply generate the same code. However, using these operators as loop indices and in conditional statements make the generated code different.

As stated in Tip #3 – Loop index, using decrementing loop index results in a smaller code size. This is also helpful to get a faster code in conditional statements.

Furthermore, pre-decrement and post-decrement also have different results. From the examples in Table 4-2, we can see that faster code is generated with a pre-decrement conditional statement. The cycle counter value here represents execution time of the longest loop.

Table 4-2. Example of conditional statement.

	Post-decrements in conditional statement	**Pre-decrements in conditional statement**

C source code	`#include <avr/io.h>`
	
	`int main(void)`
	`{`
	`  uint8_t loop_cnt = 9;`
	`  do {`
	`  if (loop_cnt--) {`
	`  PORTC ^= 0x01;`
	`  } else {`
	`  PORTB ^= 0x01;`
	`  loop_cnt = 9;`
	`  }`
	`  } while (1);`
	`}`
		`#include <avr/io.h>`
		
		`int main(void)`
		`{`
		`  uint8_t loop_cnt = 10;`
		`  do {`
		`  if (--loop_cnt) {`
		`  PORTC ^= 0x01;`
		`  } else {`
		`  PORTB ^= 0x01;`
		`  loop_cnt = 10;`
		`  }`
		`  } while (1);`
		`}`

AVR Memory Usage	Program: 104 bytes (1.3% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)	Program: 102 bytes (1.2% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Cycle counter	75	61
Compiler optimization level	-O3	-O3

The “loop_cnt” is assigned with different values in the two examples in Table 4-2 to make sure the examples work the same: PORTC0 is toggled nine times while POTRB0 is toggled once in each turn.

4.3 Tip #10 – Unrolling loops

In some cases, we could unroll loops to speed up the code execution. This is especially effective for short loops. After a loop is unrolled, there are no loop indices to be tested and fewer branches are executed each round in the loop.

The example in Table 4-3 will toggle one port pin ten times.

Table 4-3. Example of unrolling loops.

	Loops	**Unrolling loops**

C source code	`#include <avr/io.h>`
	
	`int main(void)`
	`{`
	`  uint8_t loop_cnt = 10;`
	`  do {`
	`  PORTB ^= 0x01;`
	`  } while (--loop_cnt);`
	`}`
		`#include <avr/io.h>`
		
		`int main(void)`
		`{`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`  PORTB ^= 0x01;`
		`}`

AVR Memory Usage	Program: 94 bytes (1.5% full) (.text + .data + .bootloader) Data: 0 bytes (0.1% full) (.data + .bss + .noinit)	Program: 142 bytes (1.7% full) (.text + .data + .bootloader) Data: 0 bytes (0.0% full) (.data + .bss + .noinit)
Cycle counter	80	50
Compiler optimization level	-O2	-O2

By unrolling the do { } while ( ) loop, we significantly speed up the code execution from 80 clock cycles to 50 clock cycles.

Be aware that the code size is increased from 94 bytes to 142 bytes after unrolling the loop. This is also an example to show the tradeoff between speed and size optimization.

NOTE

If -O3 option is enabled in this example, the compiler will unroll the loop automatically and generate the same code as unrolling loop manually.

4.4 Tip #11 – Control flow: If-else and switch-case

“if-else” and “switch-case” are widely used in C code; a proper organization of the branches can reduce the execution time.

For “if-else”, always put the most probable conditions in the first place. Then the following conditions are less likely to be executed. Thus time is saved for most cases.

Using “switch-case” may eliminate the drawbacks of “if-else”, because for a “switch-case”, the compiler usually generates lookup tables with index and jump to the correct place directly.

If it's hard to use "switch-case", we can divide the "if-else" branches into smaller sub-branches. This method reduces the executions for a worst case condition. In the example below, we get data from ADC and then send data through USART. "ad_result <= 240" is the worst case.

Table 4-4. Example of if-else sub-branch.

	if-else branch	**if-else sub-branch**

C source code	`#include <avr/io.h>`
	
	`uint8_t ad_result;`
	
	`uint8_t readADC() {`
	`  return ADCH;`
	`};`
	
	`void send(uint8_t data){`
	`  UDR0 = data;`
	`};`
	
	`int main(void)`
	`{`
	`  uint8_t output;`
	`  ad_result = readADC();`
	
	`  if(ad_result <= 30){`
	`  output = 0x6C;`
	`  }else if(ad_result <= 60){`
	`  output = 0x6E;`
	`  }else if(ad_result <= 90){`
	`  output = 0x68;`
	`  }else if(ad_result <= 120){`
	`  output = 0x4C;`
	`  }else if(ad_result <= 150){`
	`  output = 0x4E;`
	`  }else if(ad_result <= 180){`
	`  output = 0x48;`
	`  }else if(ad_result <= 210){`
	`  output = 0x57;`
	`  }else if(ad_result <= 240){`
	`  output = 0x45;`
	`  }`
	`  send(output);`
	`}`
		`int main(void)`
		`{`
		`  uint8_t output;`
		`  ad_result = readADC();`
		
		`  if (ad_result <= 120){`
		`  if (ad_result <= 60){`
		`  if (ad_result <= 30){`
		`  output = 0x6C;`
		`  }`
		`  else{`
		`  output = 0x6E;`
		`  }`
		`  }`
		`  else{`
		`  if (ad_result <= 90){`
		`  output = 0x68;`
		`  }`
		`  else{`
		`  output = 0x4C;`
		`  }`
		`  }`
		`  }`
		`  else{`
		`  if (ad_result <= 180){`
		`  if (ad_result <= 150){`
		`  output = 0x4E;`
		`  }`
		`  else{`
		`  output = 0x48;`
		`  }`
		`  }`
		`  else{`
		`  if (ad_result <= 210){`
		`  output = 0x57;`
		`  }`
		`  else{`
		`  output = 0x45;`
		`  }`
		`  }`
		`  }`
		`  send(output);`
		`}`

AVR Memory Usage	Program: 198 bytes (2.4% full) (.text + .data + .bootloader) Data: 1 byte (0.1% full) (.data + .bss + .noinit)	Program: 226 bytes (2.8% full) (.text + .data + .bootloader) Data: 1 byte (0.1% full) (.data + .bss + .noinit)
Cycle counter	58 (for worst case)	48 (for worst case)
Compiler optimization level	-O3	-O3

We can see it requires less time to reach the branch in the worst case. We could also note that the code size is increased. Thus we should balance the result according to specific requirement on size or speed.

5 Example application and test result

An example application is used to show the effect of tips and tricks mentioned above. Size optimization -s option is enabled in this example.

Several (not all) tips and tricks are used to optimize this example application.

In this example, one ADC channel is used to sample the input and the result is sent out through USART every five second. If the ADC result is out of range, alarm is sent out for 30 seconds before the application is locked in error state. In the rest of the main loop, the device is put in power save mode.

The speed and size optimization results of sample application before optimization and after optimization are listed in Table 5-1.

Table 5-1. Example application speed and size optimization result.

Test Items	Before optimization	After optimization	Test result
Code size	1444 bytes	630 bytes	-56.5%
Data size	25 bytes	0 bytes	-100%
Execution speed (1)	3.88ms	2.6ms	-33.0%

Note: 1. One loop including five ADC samples and one USART transmission.

6 Conclusion

In this document, we have listed some tips and tricks about C code efficiency in size and speed. Thanks to the modern C compilers, they are smart in invoking different optimization options automatically in different cases. However, no compiler knows the code better than the developer, so a good coding is always important.

As shown in the examples, optimizing one aspect may have an effect on the other. We need a balance between code size and speed based on our specific needs.

Although we have these tips and tricks for C code optimization, for a better usage of them, a good understanding of the device and compiler you are working on is quite necessary. And definitely there are other skills and methods to optimize the code efficiency in different application cases.

7 Table of contents

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Milennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
16F, Shin Osaki Kangyo Bldg.
1-6-4 Osaki Shinagawa-ku
Tokyo 104-0032
JAPAN
Tel: (+81) 3-6417-0300
Fax: (+81) 3-6417-0370

8453A-AVR-11/11