AVR 8-bit Microcontrollers

AVR42787: AVR Software User Guide

APPLICATION NOTE

Introduction

The Atmel® AVR® core is an advanced RISC architecture created to make C code run efficiently with a low memory footprint.

This document applies to tinyAVR®, megaAVR®, and XMEGA® MCUs. This document describes some frequently used functions, general means, and frequently asked questions to help new and intermediate AVR developers with developing AVR code.

Table of Contents

1. AVR 8-bit Architecture

The AVR architecture is based upon the Harvard architecture. It has separate memories and buses for program and data. This makes it possible to fetch program and data simultaneously. It has 32 8-bit fast-access General Purpose Working Registers with a single clock cycle access time. The 32 working registers is one of the keys to efficient C coding. The registers are connected to the ALU so arithmetic and logical instructions can be performed on the data in these registers. In one clock cycle, an AVR can feed data from two arbitrary registers to the ALU, perform an operation, and write back the result to the registers.

Instructions in the program memory are executed with a single level pipeline. While one instruction is being executed, the next instruction is 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.

Refer to the “AVR CPU Core” section in the respective device datasheet for more details.

2. AVR GCC and the Toolchain

GCC stands for GNU Compiler Collection. The GCC version used with the AVR is named AVR GCC.

Refer to the GNU Compiler Collection User Manual for more details.

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.

Refer to the AVR-Libc User Manual for more details.

3. I/O Header Files

I/O header files contain identifiers for all the register names and bit names for a particular processor. They must be included when registers are being used in the code.

AVR GCC has individual I/O header files for each processor. However, the actual processor type is specified as a command line flag to the compiler. (Using the -mmcu= processor flag.) This is usually done in the Makefile. This allows you to specify only a single header file for any processor type:

```
#include <avr/io.h>
```

IAR™ also allows you to specify only a single header file for any processor type:

```
#include <ioavr.h>
```

The GCC and IAR compilers know the processor type and through the single header file above, it can pull in and include the correct individual I/O header file. This has the advantage that you only have to specify one generic header file, and you can easily port your application to another processor type without having to change every file to include the new I/O header file.

Note: IAR does not always use the same register names or bit names that are used in the AVR datasheet. There may be some discrepancies between the register names found in the AVR GCC I/O header files and the IAR I/O header files.

4. Flash Variables

The C language was not designed for processors with separate memory spaces. This means that there are various non-standard ways to define a variable whose data resides in the Program Memory (Flash).

AVR GCC uses Variable Attributes to declare a variable in Program Memory:

```
int mydata[] __attribute__((__progmem__))
```

AVR-Libc also provides a convenient macro for the Variable Attribute:

```
#include <avr/pgmspace.h>
int mydata[] PROGMEM = ...
```

Note: The PROGMEM macro requires that you include <avr/pgmspace.h >. This is the normal method for defining a variable in Program Space.

IAR uses a non-standard keyword to declare a variable in Program Memory:

```

    __flash int mydata[] = ...
```

There is also a way to create a method to define variables in Program Memory that is common between the two compilers (AVR GCC and IAR). Create a header file that has these definitions:

```
#if (defined __GNUC__)
    #define FLASH_DECLARE(x) x __attribute__((__progmem__))
#elif (defined __ICCAVR__)
    #define FLASH_DECLARE(x) __flash x
#endif
```

This code snippet checks if GCC or IAR is the compiler being used and defines a macro FLASH_DECLARE(x) that will declare a variable in Program Memory using the appropriate method based on the compiler that is being used. Then you would use it as follows:

```
FLASH_DECLARE(int mydata[] = ...);
```

In AVR GCC, to read back flash data, use the pgm_read_∗() macros defined in <avr/pgmspace.h >. All Program Memory handling macros are defined there.

In IAR, flash variables can be read directly because the IAR compiler will generate LPM instruction automatically.

There is also a way to create a method to read variables in Program Memory that is common between the two compilers (AVR GCC and IAR). Create a header file that has these definitions:

```
#if (defined __GNUC__)
    #define PROGMEM_READ_BYTE(x) pgm_read_byte(x)
    #define PROGMEM_READ_WORD(x) pgm_read_word(x)
#elif (defined __ICCAVR__)
    #define PROGMEM_READ_BYTE(x) *(x)
    #define PROGMEM_READ_WORD(x) *(x)
#endif
```

5. Interrupt Service Routine

5.1. Interrupt Service Routine Declaration and Definition

The C language Standard does not specify a standard for declaring and defining Interrupt Service Routines (ISR). Different compilers have different ways of defining registers, some of which use non-standard language constructs.

AVR GCC uses the ISR macro to define an ISR. This macro requires the header file: `<avr/interrupt.h>`. In AVR GCC an ISR is defined as follows:

```
#include <avr/interrupt.h>
ISR(PCINT1_vect)
{
    //code
}
```

In IAR:

```
#pragma vector=PCINT1_vect //C90
__interrupt void handler_PCINT1_vect()
{
    // code
}
```

or

```
_Pragma("vector=PCINT1_vect") //C99
__interrupt void handler_PCINT1_vect()
{
    // code
}
```

There is also a way to create a method to define an ISR that is common between the two compilers (AVR GCC and IAR). Create a header file that has these definitions:

```
#if defined(__GNUC__)
    #include <avr/interrupt.h>
#elif defined(__ICCAVR__)
    #define __ISR(x) _Pragma(#x)
    #define ISR(vect) __ISR(vector=vect) __interrupt void handler_##vect(void)
#endif
```

This is read by the precompiler and correct code will be used depending on which compiler is being used. An ISR definition would then be common between IAR and GCC and defined as follows:

```
ISR(PCINT1_vect)
{
    //code
}
```

5.2. Variable Updated Within An Interrupt Service Routine

Variables that are changed inside ISRs need to be declared volatile. When using the optimizer, in a loop like the following one:

```
uint8_t flag;
...
ISR(SOME_vect) {
    flag = 1;
}
...
    while (flag == 0) {
        ...
    }
```

the compiler will typically access "`flag`" only once, and optimize further accesses completely away, since its code path analysis shows that nothing inside the loop could change the value of "`flag`" anyway. To tell the compiler that this variable could be changed outside the scope of its code path analysis (e. g. within an interrupt service routine), the variable needs to be declared like:

```
volatile uint8_t flag;
```

When the variable is declared volatile as above the compiler makes certain that when the variable is updated or read it will always write changes back to SRAM memory and read the variable from SRAM.

6. Calculate UART Baud Rate

Some AVR datasheets give the following formula for calculating baud rates:

```
(F_CPU/(UART_BAUD_RATE*16UL)-1UL)
```

Unfortunately the formula does not work with all combinations of clock speeds and baud rates due to integer truncation during the division operator.

When doing integer division it is usually better to round to the nearest integer, rather than to the lowest. To do this add 0.5 (i. e. half the value of the denominator) to the numerator before the division. The formula to use is then as follows.

```
((F_CPU + UART_BAUD_RATE * 8UL) / (UART_BAUD_RATE * 16UL) - 1UL)
```

This is also the way it is implemented in <util/setbaud.h >.

7. Power Management and Sleep Modes

Use of the `SLEEP` instruction can allow an application to reduce its power consumption considerably. AVR devices can be put into different sleep modes. Refer to the device datasheet for details.

There are several macros provided in this header file to actually put the device to sleep. The simplest way is to set the desired sleep mode using `set_sleep_mode()` , and then call `sleep_mode()`. This macro automatically sets the sleep enable bit, goes to sleep, and clears the sleep enable bit.

Example:

```
#include <avr/sleep.h>
...
    set_sleep_mode(<mode>);
    sleep_mode();
```

Note: Unless your purpose is to completely lock the CPU (until a hardware reset), interrupts need to be enabled before going to sleep.

Often the ISR sets a software flag or variable that is being checked, and if set, handled in the main loop. If the sleep command is used in the main loop there is a potential for a race condition to occur. In the following code there is a race condition between sleep being issued an the flag being set in the ISR.

```
#include <avr/interrupt.h>
#include <avr/sleep.h>
...
volatile bool flag = false;
...
ISR(PCINT1_vect){
    flag = true;
}
int main(){
    ...
    while(1){
        if(flag){
            flag = false;
            ...
       }
       sleep_cpu();
       ...
    }
}
```

The problem in this code comes from the fact that the ISR can happen at virtually any point in time. If the interrupt happens just after the if statement has been evaluated the device will go to sleep without doing what is required in the if statement. The actual consequence of this is dependent on the application. A way to avoid such race conditions is to disable global interrupts before checking the SW flag using the cli() command.

Example:

```
>#include <avr/interrupt.h>
#include <avr/sleep.h>
...
volatile bool flag = false;
...
ISR(PCINT1_vect){
    flag = true;
}
int main(){
    ...
    sleep_enable();
    while(1){
        cli();
        if(flag){
            flag = false;
            ...
            sei();
```

```
                sleep_cpu();
            }
            sei();
            ...
        }
}
```

This sequence ensures an atomic test of `flag` with interrupts being disabled. If the condition is met, sleep mode will be prepared, and the `SLEEP` instruction will be scheduled immediately after a `SEI` instruction. As the instruction right after the `SEI` is guaranteed to be executed before an interrupt could trigger, it is sure the device will be put to sleep. If an interrupt is pending when global interrupts are disabled the device will then jump to the ISR and continue execution after the SEI and sleep instruction. The program flow will reach the if statement and not sit waiting for a new interrupt in sleep.

Note that some AVR datasheets recommend disabling sleep immediately after waking and enabling sleep immediately before the sleep command. This recommendation is to protect against entering sleep in case the programmer has created bad pointers. It is debatable what would be the best behavior for any given application if it starts executing code from the wrong part of flash. The best type of protection is likely to use the WDT to reset the device.

Some devices have the ability to disable the Brown Out Detector (BOD) before going to sleep. This will also reduce power while sleeping. If the specific AVR device has this ability then an additional macro is defined: `sleep_bod_disable()`. This macro generates inline assembly code that will correctly implement the timed sequence for disabling the BOD before sleeping. However, there is a limited number of cycles after the BOD has been disabled that the device can be put into sleep mode, otherwise the BOD will not truly be disabled. Recommended practice is to disable the BOD (`sleep_bod_disable()`), set the interrupts (`sei()`), and then put the device to sleep (`sleep_cpu()`), as follows:

```
cli();
if (some_condition){
    sleep_bod_disable();
    sei();
    sleep_cpu();
}
sei();
```

7.1. Functions

7.1.1. void sleep_enable

`void sleep_enable(void)`

Put the device in sleep mode. How the device is brought out of sleep mode depends on the specific mode selected with the set_sleep_mode() function. See the datasheet for your device for more details.

Set the SE (sleep enable) bit.

7.1.2. void sleep_disable

`void sleep_disable(void)`

Clear the SE (sleep enable) bit.

7.1.3. void sleep_cpu

`void sleep_cpu(void)`

Put the device into sleep mode. The SE bit must be set beforehand, and it is recommended to clear it afterwards.

7.1.4. void sleep_mode

```
void sleep_mode(void)
```

Put the device into sleep mode, taking care of setting the SE bit before, and clearing it afterward.

7.1.5. void sleep_bod_disable

```
void sleep_bod_disable(void)
```

Disable BOD before going to sleep. Not available on all devices.

8. Delay Routines

The functions described here and as found in the delay.h header file are wrappers around the basic busy-wait functions from <util/delay_basic.h>. They are meant as convenience functions where actual time values can be specified rather than a number of cycles to wait for. The idea behind is that compile-time constant expressions will be eliminated by compiler optimization so floating-point expressions can be used to calculate the number of delay cycles needed based on the CPU frequency passed by the macro F_CPU.

In order for these functions to work as intended, compiler optimizations *must* be enabled, and the delay time *must* be an expression that is a known constant at compile-time. If these requirements are not met, the resulting delay will be much longer (and basically unpredictable), and applications that otherwise do not use floating-point calculations will experience severe code bloat by the floating-point library routines linked into the application.

8.1. F_CPU

The macro F_CPU specifies the CPU frequency to be considered by the delay macros. This macro is normally supplied by the environment (e.g. from within a project header, or the project's Makefile). The value 1MHz in <util/delay.h> is only provided as a "vanilla" fallback if no such user-provided definition could be found.

In terms of the delay functions, the CPU frequency can be given as a floating-point constant (e.g. 3.6864E6 for 3.6864MHz). However, the macros in <util/setbaud.h> require it to be an integer value.

8.2. void _delay_ms

`void _delay_ms(double __ms)`

The maximal possible delay is 262.14ms / F_CPU in MHz with the highest resolution. When the user request delay which exceed the maximum possible one, `_delay_ms()` provides a decreased resolution functionality. In this mode `_delay_ms()` will work with a resolution of 1/10ms, providing delays up to 6.5535 seconds (independent from the CPU frequency). The user will not be informed about decreased resolution.

If the avr-gcc toolchain has __builtin_avr_delay_cycles() support, the maximal possible delay is 4294967.295ms/ F_CPU in MHz. For values greater than the maximal possible delay, overflows results in no delay i.e., 0ms.

Conversion of `__ms` into clock cycles may not always result in an integer. By default, the clock cycles rounded up to next integer. This ensures that the user gets at least `__ms` microseconds of delay.

Alternatively, by defining the macro `__DELAY_ROUND_DOWN__`, or `__DELAY_ROUND_CLOSEST__`, before including this header file, the algorithm can be made to round down, or round to closest integer, respectively.

Note:

The implementation of `_delay_ms()` based on __builtin_avr_delay_cycles() is not backward compatible with older implementations. In order to get functionality backward compatible with previous versions, the macro "`__DELAY_BACKWARD_COMPATIBLE__`" must be defined before including this header file. Also, the backward compatible algorithm will be chosen if the code is compiled in a *freestanding environment* (GCC option `-ffreestanding`), as the math functions required for rounding are not available to the compiler then.

8.3. void _delay_us

`void _delay_us(double __us)`

The macro F_CPU is supposed to be defined to a constant defining the CPU clock frequency (in Hertz).

The maximal possible delay is 768µs/F_CPU in MHz.

If the user requests a delay greater than the maximal possible one, `_delay_us()` will automatically call `_delay_ms()` instead. The user will not be informed about this case.

If the avr-gcc toolchain has __builtin_avr_delay_cycles() support, maximal possible delay is 4294967.295µs/F_CPU in MHz. For values greater than the maximal possible delay, overflow results in no delay i.e., 0µs.

Conversion of `__us` into clock cycles may not always result in integer. By default, the clock cycles rounded up to next integer. This ensures that the user gets at least `__us` microseconds of delay.

Alternatively, by defining the macro `__DELAY_ROUND_DOWN__`, or `__DELAY_ROUND_CLOSEST__`, before including this header file, the algorithm can be made to round down, or round to closest integer, respectively.

Note:

The implementation of `_delay_us()` based on __builtin_avr_delay_cycles() is not backward compatible with older implementations. In order to get functionality backward compatible with previous versions, the macro `__DELAY_BACKWARD_COMPATIBLE__` must be defined before including this header file. Also, the backward compatible algorithm will be chosen if the code is compiled in a *freestanding environment* (GCC option `-ffreestanding`), as the math functions required for rounding are then not available to the compiler.

9. Tips and Tricks to Reduce Code Size

The example codes and testing results in this section are based on the following conditions:

1. AVR GCC 8-bit Toolchain Version: AVR_8_bit_GNU_Toolchain_3.2.1_292 (GCC version 4.5.1).
2. Target Device: ATmega88PA.

9.1. 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.

9.1.1. Tip #1 Data Types and Sizes

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

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

Table 9-1. Data Types on AVR 8-bit Microcontrollers 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 the integer data type to an 8-bit integer). The table below shows 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 9-2. Example of Different Data Types and Sizes

	Unsigned Int (16bit)	Unsigned Char (8bit)
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 8 bits, and this means that a char is sufficient. 2 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).

The difference in size will increase if the variable is manipulated more than what is done in this example. In general both arithmetic and logical manipulation of a 16-bit variables takes more cycles and space than an 8-bit variable.

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

9.1.2. Tip #2 Global Variables and Local Variables

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 SRAM will be assigned to this variable at program link time. Access 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.

The table below shows the effect of global and local variables.

Table 9-3. 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 bytes (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 1 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.

9.1.3. 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 0 to the maximum value (increment), but it is more efficient to count the loop from the maximum value to 0 (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.

Table 9-4 shows the effect of "`do { } while ( )`" loop with increment and decrement loop index.

Table 9-4. 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--;`[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)

Note:

1. 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.

9.1.4. 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. 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. The table below shows the effect of loop jamming.

Table 9-5. 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)

9.1.5. 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). By 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 retrieve them from program space, AVR-Libc provides a simple macro "`PROGMEM`" and a macro "`pgm_read_byte`". The "`PROGMEM`" macro and "`pgm_read_byte`" macro are defined in the <avr/pgmspace.h> system header file.

The table below shows how we save SRAM by moving the global string into program space.

Table 9-6. 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, size is reduced 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. Refer to the AVR-Libc User Manual for more details.

9.1.6. Tip #6 Access Types: Static

For global data, use the keyword "`static`" 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 in most cases not be declared 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 for the compiler 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 created to jump in and out of it. The table below shows the effect of static function.

Table 9-7. 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.

9.1.7. 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. Use macros instead of functions for tasks that generates less than 2-3 lines assembly code. The table below shows the code usage of assembly macro compared with using a function.

Table 9-8. 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)

Refer to AVR-Libc User Manual for more details.

9.2. Tips and Tricks to Reduce Execution Time

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

9.2.1. 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 AVR 8-bit, accessing 8-bit (Byte) value is always the most efficient way.

The table below shows the difference between 8-bit and 16-bit variables.

Table 9-9. 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.

9.2.2. Tip #9 Conditional Statement

Usually pre-decrement and post-decrement (or pre-increment and post-increment) 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, using decrementing loop index results in a smaller code size. This is also helpful to get faster execution in conditional statements.

Furthermore, pre-decrement and post-decrement also have different results. From the table below, 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 9-10. 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 to make sure the examples work the same: PORTC0 is toggled nine times while POTRB0 is toggled once in each turn.

9.2.3. 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 table below shows the effect of unrolling loops.

Table 9-11. Example of Unrolling Loops

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

9.2.4. 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 execution time for a worst case condition. In the table below, we get data from ADC and then send data through USART. "`ad_result <= 240`" is the worst case.

Table 9-12. Example of If-Else Sub-Branch

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

	If-Else branch	If-Else sub-branch
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.

9.3. Conclusion

In this chapter, we list 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.

10. References

- GNU Compiler Collection Manual (https://gcc.gnu.org/onlinedocs/gcc/)
- AVR-Libc User Manual (http://www.nongnu.org/avr-libc/user-manual/)
- Atmel Studio (http://www.atmel.com/tools/atmelstudio.aspx?tab=overview)
- Atmel START (http://start.atmel.com)
- IAR C/C++ Compiler User Guide for AVR (http://ftp.iar.se/WWWfiles/AVR/webic/doc/EWAVR_CompilerReference.pdf)

11. Revision History

Doc Rev.	Date	Comments
42787A	10/2016	Initial document release.