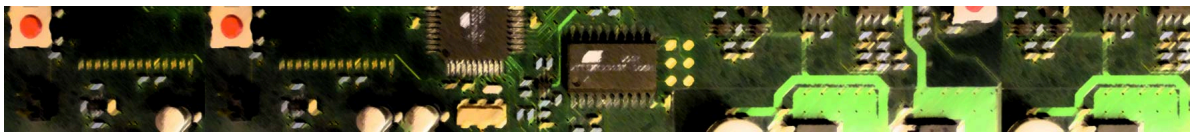

(/)

Разделы

Новости (/index.php/novosti.html)
Встраиваемые системы (/index.php/embedded-programming.html)
Программирование AVR (/index.php/programming-avr.html)
Программирование ARM (/index.php/programmirovanie-arm.html)
Инструменты/ технологии (/index.php/instruments-technologies.html)
Как подключить (/index.php/how-connection.html)
Компоненты (/index.php/electronic-components.html)
RTOS (/index.php/rtos.html)
Софт (/index.php/iar-embedded-workbench.html)
Проекты (/index.php/projects-avr.html)
Ссылки (/index.php/links.html)

AVR4027: Трюки и советы по оптимизации Си кода для 8-и разрядных AVR микроконтроллеров. Ч.1

13/09/2013 - 18:42 | Павел Бобков

Особенности

- Введение в ядро Atmel AVR и Atmel AVR GCC
- Советы и трюки по уменьшению размера кода
- Советы и трюки по уменьшению времени выполнения кода
- Примеры применения

1. Введение

AVR ядро основано на продвинутой RISC архитектуре оптимизированной для Си кода. Это позволяет разрабатывать хорошие и дешевые продукты с широкой функциональностью.

Когда речь идет об оптимизации, мы обычно имеем в виду две вещи: размер кода и скорость его выполнения. В настоящее время Си компиляторы имеют различные варианты оптимизации, позволяющие разработчикам получать эффективный код по одному из этих критериев.

Хороший Си код дает компилятору больше возможности по его оптимизации. Однако, в некоторых случаях оптимизация кода по одному из критериев ухудшает другой, поэтому разработчик должен искать баланс между ними для удовлетворения своих требований. Понимание некоторых нюансов программирования на Си для AVR позволяет разработчикам фокусировать свои усилия в нужном направлении для достижения эффективного кода.

В этой статье мы рассмотрим рекомендации по программированию на Си для компилятора avr-gcc. Однако эти советы могут быть использованы и с другими компиляторами.

2. Atmel AVR ядро и Atmel AVR GCC

Прежде чем оптимизировать программное обеспечение, необходимо иметь хорошее понимание устройства AVR ядра и стратегий AVR GCC, которые используются для

2.1 Архитектура 8-и разрядного AVR

AVR использует Гарвардскую архитектуру с раздельной памятью и шиной для программы и данных. Он имеет регистровый файл из 32 8-и разрядных рабочих регистров общего назначения с временем доступа один тактовый цикл. 32 рабочих регистра – один из ключей к эффективному Си программированию. Эти регистры имеют такие же функции, как и традиционные аккумуляторы, только их 32 штуки. За один такт AVR может передать два произвольных регистра арифметическому логическому устройству, выполнить операцию и записать результат обратно в регистровый файл.

Инструкции в памяти программ выполняются на одном конвейерном уровне. Пока одна команда выполняется, следующая извлекается из памяти. Эта концепция позволяет выполнять команды за один такт. Большинство инструкций AVR имеют 16-и разрядный формат. Каждый адрес памяти программ содержит 16 или 32 разрядные инструкции.

Для более детальной информации почитайте раздел “AVR CPU Core” в документации на микроконтроллеры.

2.2 AVR GCC

GCC расшифровывается как коллекция GNU компиляторов (GNU Compiler Collection). GCC используемый для AVR микроконтроллеров называется AVR GCC.

AVR GCC имеет несколько уровней оптимизации: -O0, -O1, -O2, -O3 и -Os. Для каждого уровня разрешены свои параметры. Исключение составляет уровень -O0 – для него оптимизация полностью отключена. Кроме разрешенных для каждого уровня опций, можно также включать отдельные параметры оптимизации, чтобы получить специфическую обработку кода.

Для ознакомления с полным списком параметров и уровней оптимизации обратитесь к руководству на GNU компиляторы. Руководство можно найти по ссылке ниже.

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

Кроме компилятора, AVR GCC включает в себя другие инструменты, которые работают вместе для получения конечного исполняемого приложения для AVR микроконтроллера. Эта группа инструментов называется “тулчейном “ (toolchain). Важную функцию в этом AVR тулчейне выполняет AVR-Libc, которая обеспечивает функции стандартной Си библиотеки, а также множество дополнительных специфичных для AVR функций и базовым стартовым кодом (startup code). Руководство на AVR Libc вы можете найти по ссылке ниже.

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

2.3 Платформа разработки

Все примеры из этого документа тестировались с использованием следующих инструментов:

1wire (/index.php/programming-avr/tag/1wire.html)
arm (/index.php/programming-avr/tag/arm.html)
avr (/index.php/programming-avr/tag/avr.html)
avr программатор (/index.php/programming-avr/tag/avr программатор.html)
ds18b20 (/index.php/programming-avr/tag/ds18b20.html)
eeprom (/index.php/programming-avr/tag/eeprom.html)
i2c (/index.php/programming-avr/tag/i2c.html)
IAR (/index.php/programming-avr/tag/IAR.html)
lcd (/index.php/programming-avr/tag/lcd.html)
tsop (/index.php/programming-avr/tag/tsop.html)
twi (/index.php/programming-avr/tag/twi.html)
алгоритмы (/index.php/programming-avr/tag/алгоритмы.html)
библиотеки (/index.php/programming-avr/tag/библиотеки.html)
датчик (/index.php/programming-avr/tag/датчик.html)
драйвер (/index.php/programming-avr/tag/драйвер.html)
интерфейс (/index.php/programming-avr/tag/интерфейс.html)
компоненты (/index.php/programming-avr/tag/компоненты.html)
макросы (/index.php/programming-avr/tag/макросы.html)
оу

3. Советы и трюки по уменьшению размера кода

В этом разделы мы перечислим некоторые рекомендации по уменьшению размера кода. Для каждого совета будет дано описание и пример.

3.1 Совет #1 - типы данных и размеры

Используйте наименьший тип данных, способный представить вашу переменную. Для чтения 8-и разрядного регистра, например, не нужно использовать 2-х байтную переменную, достаточно однобайтной.

Размеры типов данных для AVR можно посмотреть в заголовочном файле stdint.h и в таблице 3-1.

Таблица 3-1. Типы данных для AVR, описанные в stdint.h

Типы данных		Размер
signed/unsigned char	int8_t/uint8_t	8 бит
signed/unsigned int	int16_t/uint16_t	16 бит
signed/unsigned long	int32_t/uint32_t	32 бита
signed/unsigned long long	int64_t/uint64_t	64 бита

Имейте в виду, что некоторые опции компилятора могут повлиять на типы данных (например, AVR-GCC имеет опцию –mint8, которая делает тип int 8-и разрядным).

Два примера кода в таблице 3-2 показывают результат использования разных типов данных. Также в таблице показаны размеры кода, получаемого при построении проекта с опцией оптимизации –Os (оптимизация для размера).

Таблица 3-2. Пример использования различных типов данных.

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

Си код	#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();
		}

Использование памяти AVR	Program: 92 bytes (1.1% full)	Program: 90 bytes (1.1% full)
Опция оптимизации	-Os (optimize for size)	-Os (optimize for size)

В левом примере мы пользуемся 2-х байтным типом данных для временной переменной и возвращаемого значения. В правом примере вместо этого используется

avr/tag/программирование микроконтроллеров.html)

расчет (/index.php/programming-avr/tag/расчет.html)

семисегментный индикатор (/index.php/programming-avr/tag/семисегментный индикатор.html)

си (/index.php/programming-avr/tag/си.html)

событийная система (/index.php/programming-avr/tag/событийная система.html)

схемотехника (/index.php/programming-avr/tag/схемотехника.html)

таймер (/index.php/programming-avr/tag/таймер.html)

управление (/index.php/programming-avr/tag/управление.html)

устройства (/index.php/programming-avr/tag/устройства.html)

учебный курс avr (/index.php/programming-avr/tag/учебный курс avr.html)

шим (/index.php/programming-avr/tag/шим.html)

3.2 Совет #2 - глобальные переменные и локальные значения

В большинстве случаев не рекомендуется использовать глобальные переменные. Применяйте локальные переменные везде, где возможно. Если переменная используется только в функции, ее следует объявлять внутри функции как локальную переменную.

Теоретически, выбор между глобальными и локальными переменными должен определяться тем, как она используется.

Если переменная объявлена как глобальная, в оперативной памяти для нее выделяется уникальный адрес. Также для доступа к глобальной переменной, как правило, используются дополнительные байты (по 2 на 16-и разрядный адрес), чтобы получить ее адрес.

Локальные переменные обычно размещаются в регистрах или в стеке. Когда вызывается функция, локальные переменные задействуются. Когда функция завершает свою работу, локальные переменные могут быть удалены.

Два примера в таблице 3-3 показывают эффект применения глобальных и локальных переменных.

Таблица 3-3. Пример глобальных и локальных переменных.

	Глобальные переменные	Локальные переменные

Си код	#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;
		}

Использование памяти AVR	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)
Опция оптимизации	-Os (optimize for size)	-Os (optimize for size)

В левом примере мы объявляем однобайтовую глобальную переменную. Avr-size утилита показывает, что мы используем 104 байта памяти программ и один байт памяти данных при оптимизации –Os.

В правом примере, мы объявляем локальную переменную внутри main() функции и код уменьшается до 84 байтов, а оперативная память не используется совсем.

3.3 Совет #3 - индекс цикла

Циклы широко используются при программировании микроконтроллеров. В Си существуют три типа циклов – “while(){}”, “for(){}“ и “do{}while();”. Если включена опция

Однако более эффективно делать наоборот – считать от максимального значения до нуля, то есть использовать декремент.

При инкременте, в каждой итерации цикла необходимо выполнять инструкцию сравнения индексной переменной с максимальным значением. Когда мы используем декремент, в этой инструкции нет необходимости. Как только индексная переменная достигнет нуля, в регистре SREG установится флаг Z.

В таблице 3-4 приведены примеры кода, использующего цикл “do {}while()” с инкрементом и декрементом индексной переменной.

Таблица 3-4. Пример do{}while() циклов с инкрементом и декрементом индексной переменной

	do{}while() с инкрементом индексной переменной	do{}while() с декрементом индексной переменной

Си код	#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);
		
		}

Использование памяти AVR	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)
Опция оптимизации	-Os (optimize for size)	-Os (optimize for size)

Чтобы иметь более прозрачный Си код, мы записали этот пример как “do{count--;}while(count);”, а не “do{}while(count--);”, как обычно пишут в книгах по Си. Однако в обоих случаях размер кода будет одинаковым.

3.4 Совет #4 - объединение циклов

Иногда циклы реализованы однотипно и это может приводить к длинному списку итераций. Объединив выражения и операторы этих циклов в один, мы можем уменьшить общее количество циклов в коде. При этом уменьшится как размер кода, так и время его выполнения. В таблице 3-5 вы можете видеть пример использования объединения циклов.

Таблица 3-5. Пример объединения циклов.

	Раздельные циклы	Объединенные циклы

Си код	#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;
		
		}

Использование памяти AVR	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)
Опция оптимизации	-Os (optimize for size)	-Os (optimize for size)

3.5 Совет #5 – константы в программной памяти

Глобальные переменные, таблицы и массивы, которые никогда не меняются, должны размещаться во флэш памяти микроконтролера. Это позволяет экономить оперативную память.

В этом примере мы не используем ключевое слово “const”. Потому что объявление переменной с “const” сообщает компилятору, что ее значение не будет меняться, но не определяет место хранения.

Таблица 3-6. Пример констант в программной памяти.

	Константы в памяти данных	Константы в памяти программ

Си код	#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]);
		}

Использование памяти AVR	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)
Опция оптимизации	-Os (optimize for size)	-Os (optimize for size)

После того как мы поместили константы в программную память, уменьшилось и ее использование и ОЗУ. Однако появились небольшие накладные расходы связанные с чтением данных, так как чтение из флэш памяти выполняется медленнее, чем из оперативной памяти.

Если данные, хранящиеся во флэш памяти, используются в коде несколько раз, мы можем получить меньший код, используя дополнительную временную переменную вместо многократного вызова макроса “pgm_read_byte”.

В заголовочном файле pgmspace.h есть несколько макросов и функций для сохранения и чтения различных типов данных в/из памяти программ. Для большей информации обратитесь к руководству на avr-libc.

3.6 Совет #6 – типы доступа: static

Для глобальных данных использование ключевого слова static не всегда возможно. Если глобальные переменные объявлены с ключевым словом static, они могут быть доступны только в том файле, внутри которого они определены. Это предотвращает случайное использование переменной в других файлов.

С другой стороны, объявления локальных переменных внутри функции с ключевым словом static следует избегать. Значение статической переменной сохраняется между вызовами функции, и эта переменная сохраняется в течение всей программы. Таким образом, она требует постоянного места хранения в ОЗУ и дополнительного кода для доступа к ней. Это похоже на глобальную переменную, только область видимости статической переменной ограничена телом функции, в котором она объявлена.

Статическими можно объявлять также функции. В этом случае область видимости функции будет тоже ограничена файлом, в котором она объявлена, и вызвать ее можно будет только оттуда. По этим причинам компилятору проще оптимизировать такие функции.

Если статическая функция вызывается в файле всего один раз и оптимизация разрешена (-O1, -O2, -O3, -Os), компилятор сделает эту функцию встраиваемой.

Таблица 3-7. Пример использования типов доступа – статическая функция.

	Глобальная функция (вызывается однократно)	Статическая функция (вызывается однократно)

Си код	#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;
		}

Использование памяти AVR	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)
Опция оптимизации	-Os (optimize for size)	-Os (optimize for size)

Заметьте, если функция вызывается несколько раз, она не будет встраиваться, потому что это увеличит код больше, чем прямой вызов функции.

3.7 Совет #7 – низко уровневые ассемблерные инструкции

Ассемблерные команды всегда лучше оптимизированного кода. Единственный недостаток ассемблерного кода – это непереносимый синтаксис, так что это в большинстве случаев его не рекомендуется использовать.

Чтобы улучшить читаемость и портируемость кода, можно использовать ассемблерные макросы. Такими макросами можно заменять функции, которые генерируются в 2-3 ассемблерные строки. В таблице 3-8 показан пример использования ассемблерного макроса вместо функции.

Таблица 3-8. Пример использования низкоуровневых ассемблерных инструкций.

	Функция	Ассемблерный макрос

Си код	#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){
		  }
		}

Использование памяти AVR	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)
Опция оптимизации	-Os (optimize for size)	-Os (optimize for size)

Для более подробной информации относительно использования ассемблера с языком Си на AVR, ознакомьтесь с разделом “Inline Assembler Cookbook” в руководстве на avr-libc.

микроконтроллеров. Ч.2 (/index.php/programming-avr/item/180-avr4027-tips-and-tricks-to-optimize-your-c-code.html)
Проект к статье - avr4027.zip (http://www.atmel.com/Images/AVR4027.zip)

Вольный перевод - ChipEnable.Ru (http://ChipEnable.Ru)

Tweet (https://twitter.com/share)

Tagged under #avr (/index.php/programming-avr/tag/avr.html) #си (/index.php/programming-avr/tag/си.html) #оптимизация (/index.php/programming-avr/tag/оптимизация.html)

Related items

- Библиотека для опроса кнопок (/index.php/programming-avr/218-biblioteka-dlya-oprosa-knopok.html)
- Работа с SD картой. Воспроизведение wav файла. Ч3 (/index.php/programming-avr/212-rabota-s-sd-kartoy-vosproizvedenie-wav-fayla-ch3.html)
- Работа с SD картой. Подключение к микроконтроллеру. Ч1 (/index.php/programming-avr/209-rabota-s-sd-kartoy-podklyuchenie-k-mikrokontrolleru-ch1.html)
- AVR315: Использование TWI модуля в качестве ведущего I2C устройства (/index.php/programming-avr/208-avr315-ispolzovanie-twi-modulya-v-kachestve-veduschego-i2c-ustroystva.html)
- ATtiny10. Самый маленький микроконтроллер AVR (/index.php/programming-avr/200-attiny10-samyy-malenkiy-mikrokontroller-avr.html)

Comments (/index.php/component/jcomments/feed/com_k2/179.html)

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3236) **Bonio** 2013-09-14 20:24
Спасибо, очень полезная информация.

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3245) **megannnn98** 2013-09-16 04:51
Спасибо, только немного трудно читать - чувствуется что перевод. :zzz

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3254) **Pashgan** 2013-09-16 19:16
Ученые (в моем лице) постоянно бьются над улучшением качества переводов.

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3262) **САБ** 2013-09-18 09:30
Совет №7 очень и очень спорный, особенно из-за слова "всегда". А уж пример вообще ни о чем -
1) выигрыш получился за счет встраивания, а не замены на асм. Такой же эффект можно было получить, объявив функцию static, ну в крайнем случае static inline.
2) сама ассемблерная вставка написана с ошибкой - используемый регистр r24 не указан в списке параметров вставки, значит эта вставка испортит предыдущее содержимое этого регистра и в чуть более сложном коде это приведет к неправильному поведению программы
3) выбор регистра надо было доверить компилятору через использование параметров асм-вставки.

В общем от этого совета больше вреда, чем пользы.

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3267) **Pashgan** 2013-09-18 11:48
Да, совет более чем сомнительный. Я бы его не стал использовать.

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3271) **FreshMan** 2013-09-19 08:31
вопрос по поводу констант
допустим есть
constant char x;
char y;
насколько я понимаю то для обоих выделитсяминимальная ячейка памяти в 8 бит
где же тогда сдесь будет оптимизация ?

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3272) **САБ** 2013-09-19 09:59
Если вы о константах в программной памяти, то экономия вот в чем - константная переменая в ОЗУ должна откуда-то получить свое значение. И это начальное значение хранится во флеше, а перед запуском main()

Для хранения значений во флеш используется атрибут PROGMEM, а const лишь указывает компилятору, что переменная не может изменить свое значения.
const никак не влияет на потребляемые ресурсы, всё отличие от обычной переменной в том, что запись такого вида вызовет ошибку компиляции.
const char x=1;
x = 2;

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3274) **FreshMan** 2013-09-19 11:49
но ведь "переменная" с префиксом const тоже хранится во флеше и тоже грузится в ОЗП
в чем же разница...., не пойму.....

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3275) **_Артём_** 2013-09-19 12:04
Разница в том что при использовании flash переменной её копия в ОЗУ не создаётся, а значение считывается только тогда когда нужно.

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3276) **Bonio** 2013-09-19 12:11
Нет, переменная const так же храниться в оперативке. Просто компилятор следит за тем, чтобы в программе нигде не менялось её значение.
Для хранения значения во флеш используйте атрибут PROGMEM, для чтения такого значения используйте процедуру eeprom_read_byte.

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3277) **Bonio** 2013-09-19 12:12
Тьфу, pgm_read_byte конечно же.

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3278) **FreshMan** 2013-09-19 15:29
а почему ТОЛЬКО в цикле “do{}while()” инкремент и декремент индексной переменной цикла будет давать код разного размер ?
в не цикла инкремент и декремент выполняются выходит за одинаковое время ?

(/index.php/programming-avr/item/179-avr4027-tips-and-tricks-to-optimize-your-c-code.html#comment-3279) **Pashgan** 2013-09-19 18:57
Если используется инкремент, то в каждом цикле выполняется команда сравнения, а затем команда перехода.

Например:

Code:

```
LOOP:
...

inc r16
cpi r16, 10
brcs LOOP
```

При декременте выполняется только команда перехода.

Например:

Code:

```
LOOP:
...
dec r16
brne LOOP
```

Вне цикла операции инкремента и декремента выполняются за одинаковое время. Это ассемблерные инструкции inc и dec.