

# C++ для «самых маленьких»

## Содержание

Введение .....	2
1 Особенности использования языка C++ в среде IAR Embedded Workbench .....	3
1.1 Использование ключевого слова inline .....	3
1.2 Использование директивы #pragma vector.....	3
1.3 Настройка проекта в IAR Embedded Workbench for Atmel AVR.....	3
2 Принцип построения программ на языке C++ .....	9
2.1 Библиотека MCU.....	9
2.2 Флаги.....	9
2.3 Прерывания.....	13
2.4 Драйвера, устройства.....	14
2.5 Периферия микроконтроллеров AVR .....	16
2.5.1 Сторожевой таймер (WDT).....	16
2.5.2 Таймеры (Timer0).....	17
2.6 Результаты компиляции .....	19
3 Пример программы на языке C++ (динамическая индикация) .....	20
3.1 Описание .....	20
3.2 Настройка периферии микроконтроллера .....	21
3.3 Счётчик .....	22
3.4 Зуммер.....	23
3.5 Клавиатура .....	23
3.6 7-сегментный индикатор .....	24
3.7 Прерывания.....	25
3.8 Основной цикл программы .....	26
3.9 Результаты компиляции проекта .....	27
4 Пример программы на языке C++ (символьный LCD-индикатор на контроллере HD44780) ....	28
4.1 Описание .....	28
4.2 Драйвер «LCD-индикатор» .....	29
4.3 Устройство «LCD-индикатор».....	30
4.4 Работа с буфером индикатора.....	31
4.5 Настройка и использование .....	32
4.6 Конвертирование текста.....	33
4.7 Результаты компиляции проекта.....	34

## Введение

Речь пойдёт об использовании языка C++ для микроконтроллеров младших семейств (с небольшим объёмом памяти).

Основная причина использования языка C++ при разработке устройств на микроконтроллерах, это попытка перевести выполнение части кода на этап компиляции, тем самым освободив разработчика от ненужных расчётов и дублирования кода, а микроконтроллер от ненужной работы. При этом код будет представлять собой законченный модуль – класс.

Из всего многообразия возможностей языка C++ для микроконтроллеров смело можно использовать следующие:

- классы со статическими методами и переменными (без конструктора);
- перегрузка методов;
- перегрузка операторов;
- наследование;
- шаблоны;
- шаблонное метапрограммирование.

Какие преимущества даёт использование языка C++ для микроконтроллеров:

- использование классов позволяет лучше структурировать программу;
- перегрузка методов позволяет улучшить читаемость кода за счёт уменьшения разнообразия имён методов (функций);
- перегрузка операторов позволяет производить обычные арифметические действия над собственными типами данных;
- использование шаблонов позволяет избавиться от многократного повторения кода при использовании различных типов данных;
- шаблонное метапрограммирование позволяют по новому взглянуть на использование языка C++ для микроконтроллеров. Большая работа в этом направлении проделана Константином Чижовым (Neiver) и описана в статье «Работа с портами ввода/вывода микроконтроллеров на Си++». Его наработки строятся на использовании библиотеки loki, автором которой является Андрей Александреску. Более подробно о шаблонном метапрограммировании можно прочитать в книге «Андрей Александреску. Современное проектирование на C++».

Какие трудности возникают при переходе на язык C++ для микроконтроллеров:

- для выбранного микроконтроллера может не существовать компилятора C++;
- трудности при отладке;
- сообщения об ошибках не информативны.

Существует ещё одна сложность разработки (поддержки) программ на языке C++ для микроконтроллеров. Дело в том, что каждый разработчик использует свой подход при работе с аппаратной частью микроконтроллера. Поэтому переносимость кода между различными проектами (разработчиками) тормозится стилем работы с прерываниями, портами ввода/вывода, а также другой периферией микроконтроллера.

При аккуратном использовании языка C++ исполняемый код по размеру получается соизмерим с аналогичным кодом, написанным на языке Си. Размер используемой RAM зависит только от разработчика. Если объём RAM позволяет небольшое расточительство, то вполне можно этим воспользоваться, создав более «красивые» классы.

В сети Internet встречается несколько вариантов построения программ на языке C++. Хочу предложить ещё один вариант.

Для примера выберем микроконтроллер ATtiny2313 (также будем проверять код на ATmega32) компании Atmel ([www.atmel.com](http://www.atmel.com)). В качестве среды разработки будем использовать IAR Embedded Workbench for Atmel AVR 6.80 компании IAR ([www.iar.com](http://www.iar.com)), чьи компиляторы являются одними из лучших для микроконтроллеров различных архитектур.

При переходе на язык C++ я старался не отходить далеко от стандартного языка Си. Как говориться «без фанатизма».

## 1 Особенности использования языка C++ в среде IAR Embedded Workbench

### 1.1 Использование ключевого слова inline

Для получения компактного кода на языке C++ в среде IAR Embedded Workbench будем активно использовать ключевое слово `inline`, которое позволяет встроить код без использования вызова функции (метода). Пример использования ключевого слова `inline`:

```
1 #pragma inline = forced
2 static inline void Init(const uint8_t value)
3 {
4     Data = value ? 0xFF : 0;
5 }
```

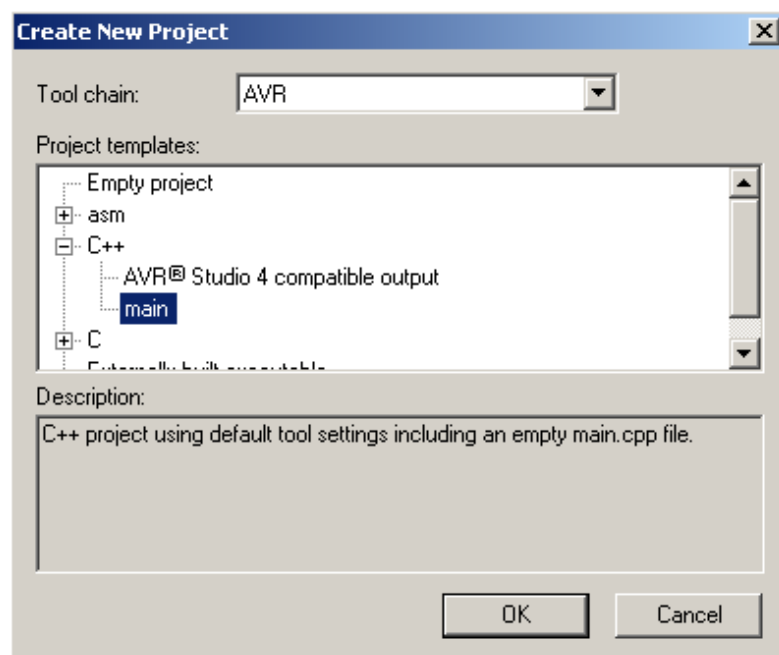
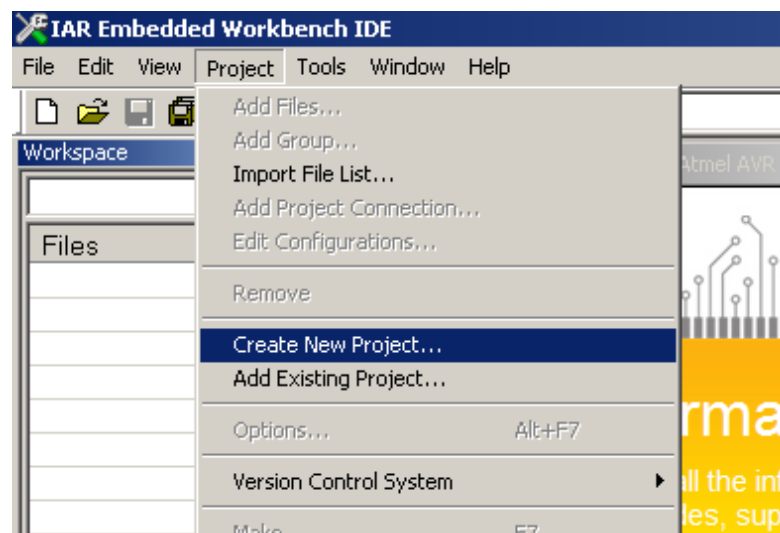
Строка 1 сообщает компилятору о необходимости принудительно использовать ключевое слово `inline`. Без данной директивы компилятор по своему усмотрению будет обрабатывать ключевое слово `inline`.

### 1.2 Использование директивы #pragma vector

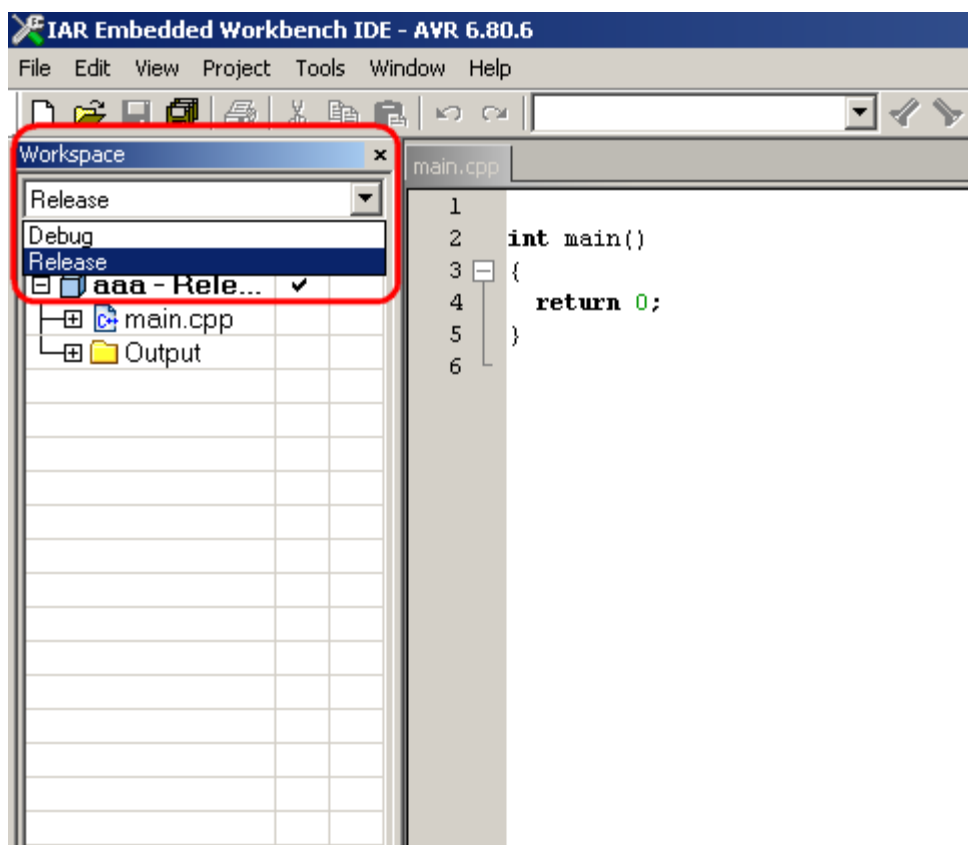
Адреса векторов прерываний будем задавать в шестнадцатеричном виде, т.к. от версии к версии среды разработки IAR Embedded Workbench имена векторов прерываний меняются.

### 1.3 Настройка проекта в IAR Embedded Workbench for Atmel AVR

Создаём новый проект.

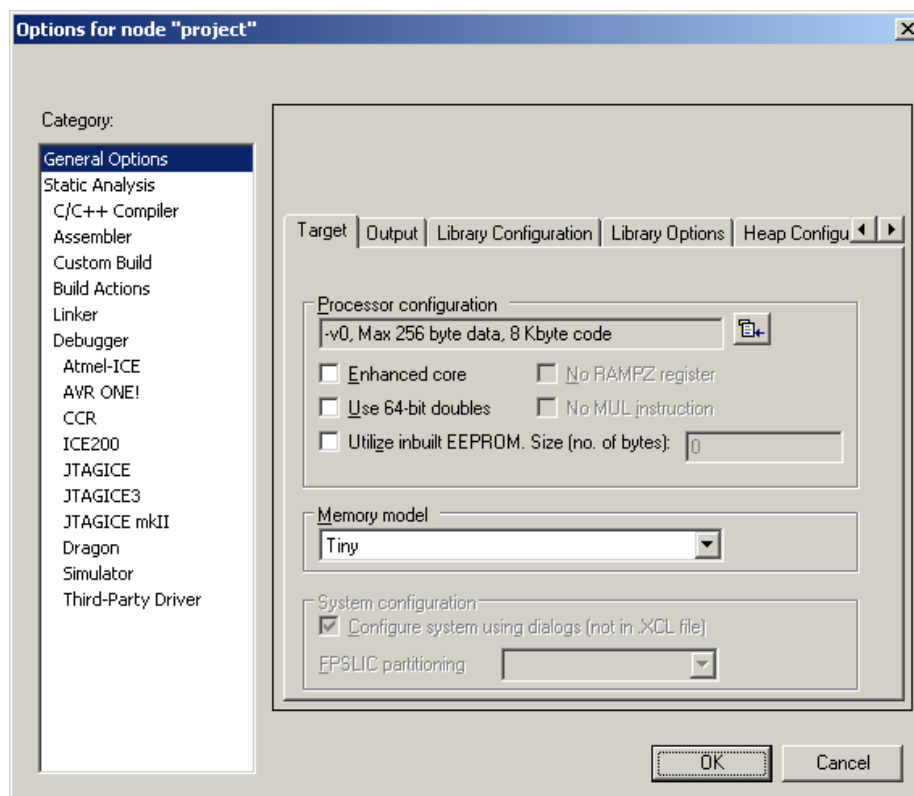


Выбираем конфигурацию «Release».



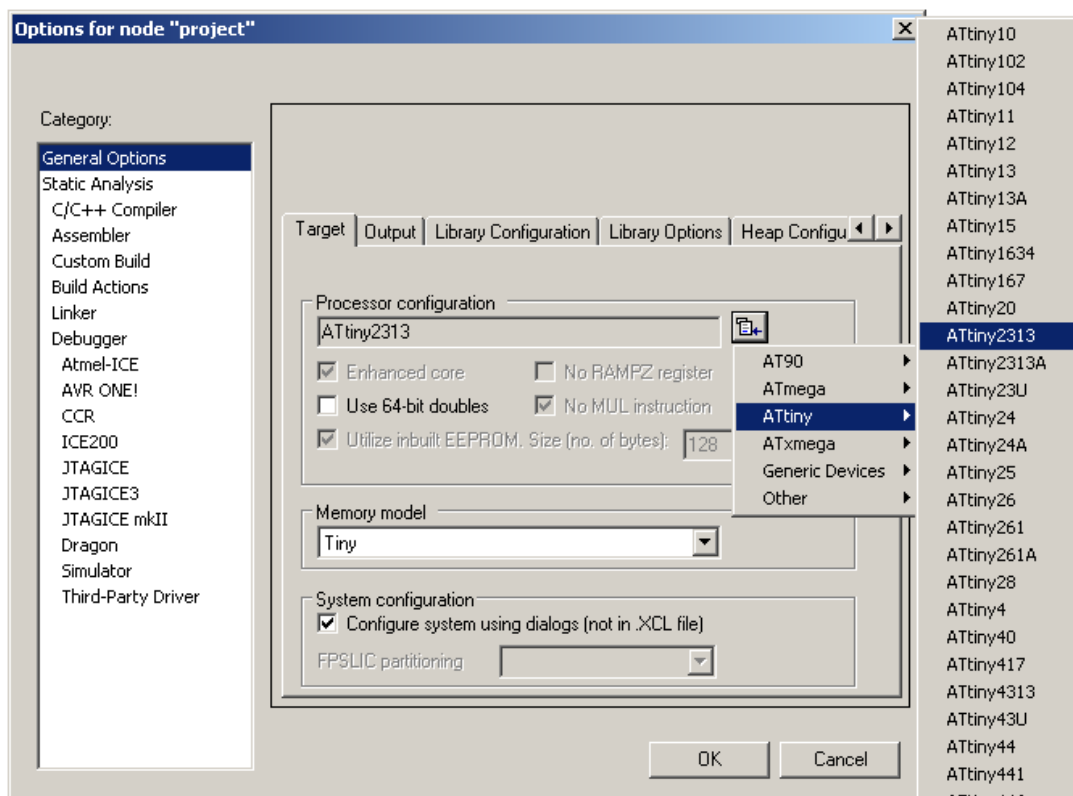
Вызов окна настроек проекта.

Для отображения окна настроек нажимаем Alt-F7. Данные параметры будут относиться к конфигурации «Release», для конфигурации «Debug» настройку параметров придётся повторить.

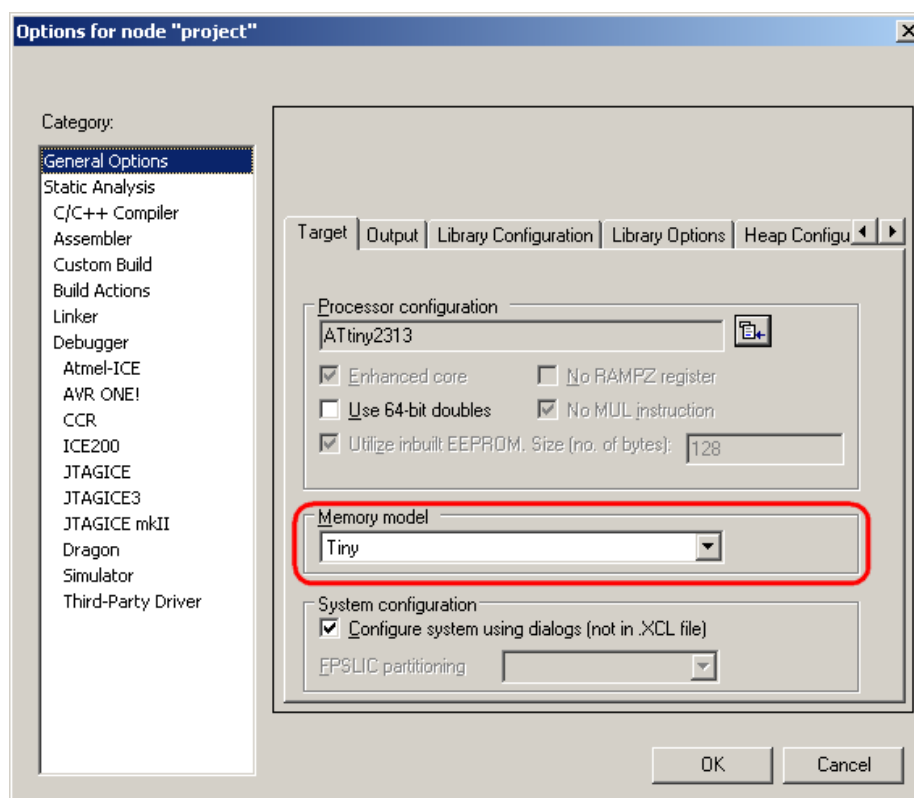


Установка типа микроконтроллера.

Выбираем микроконтроллер ATtiny2313.

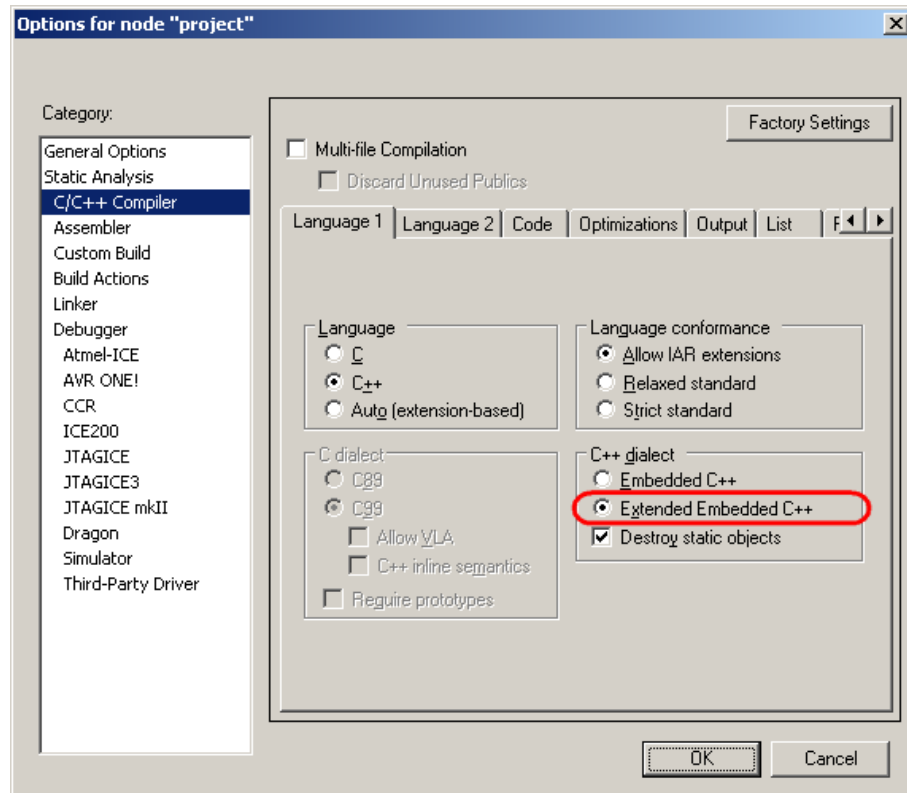
Выбираем модель памяти.

Для микроконтроллера ATtiny2313 предлагается только модель «Tiny».



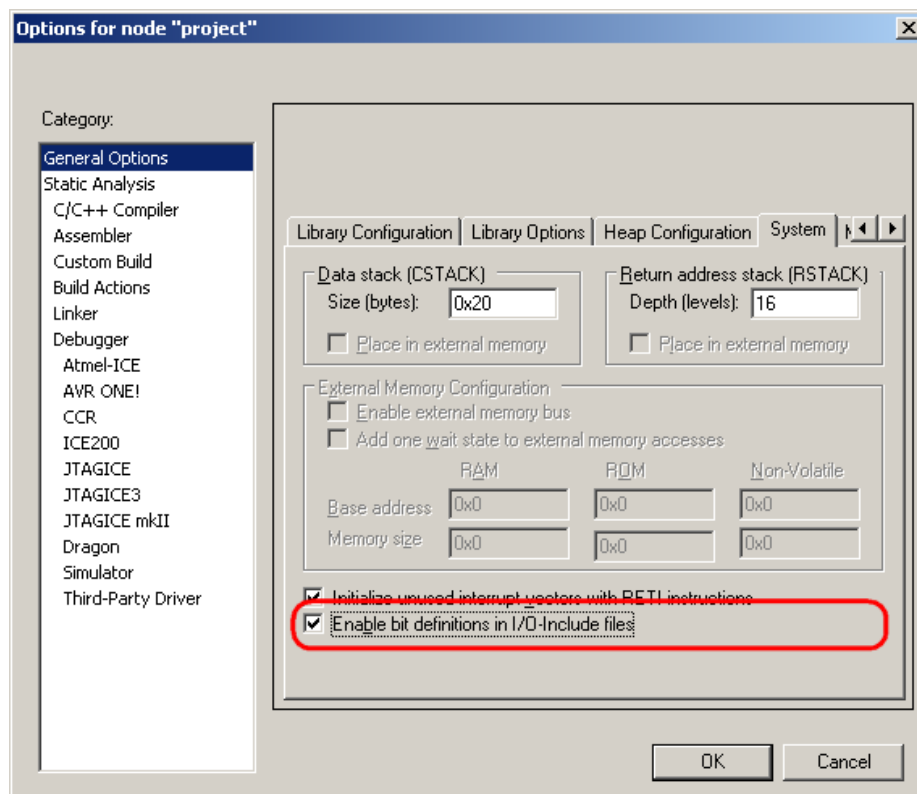
### Выбор диалекта C++.

Выбираем расширенную версию диалекта языка C++ для поддержки шаблонов.



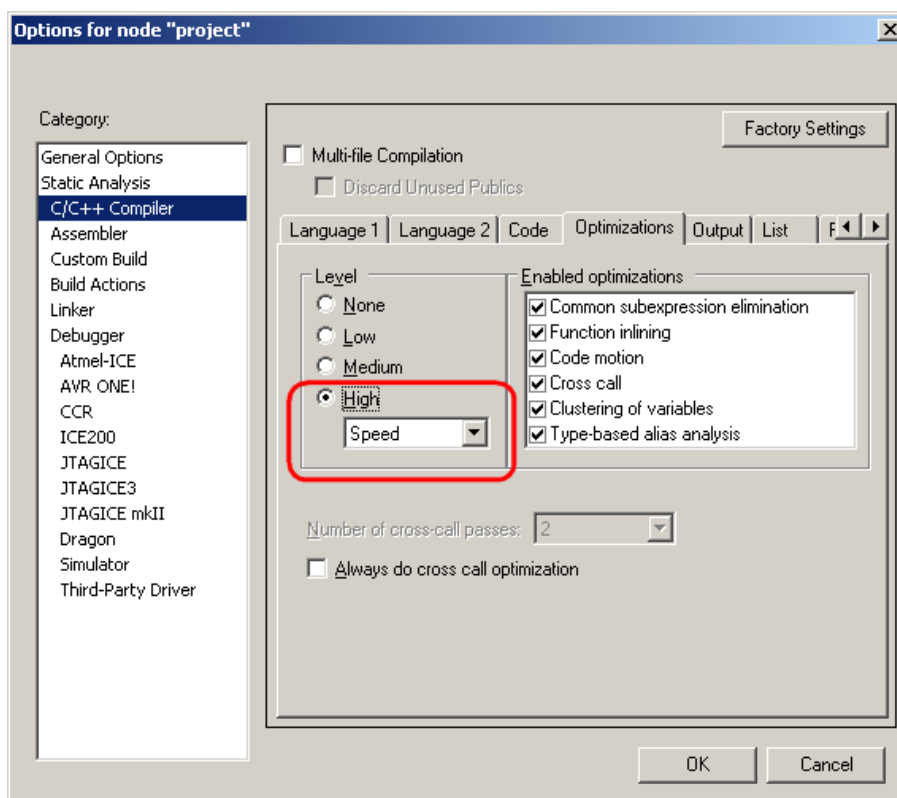
### Настройка системных параметров.

Включаем опцию «Enable bit definitions in I/O-Include files» для использования заголовочных файлов данного микроконтроллера с описанием регистров ввода/вывода.



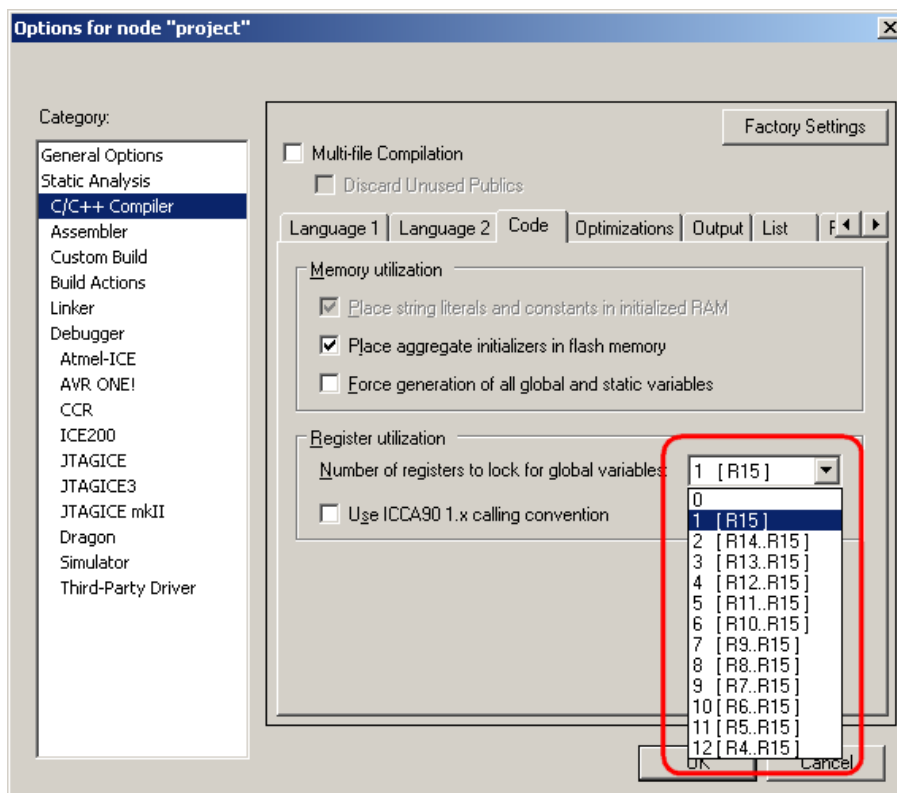
### Настройка оптимизации компилятора.

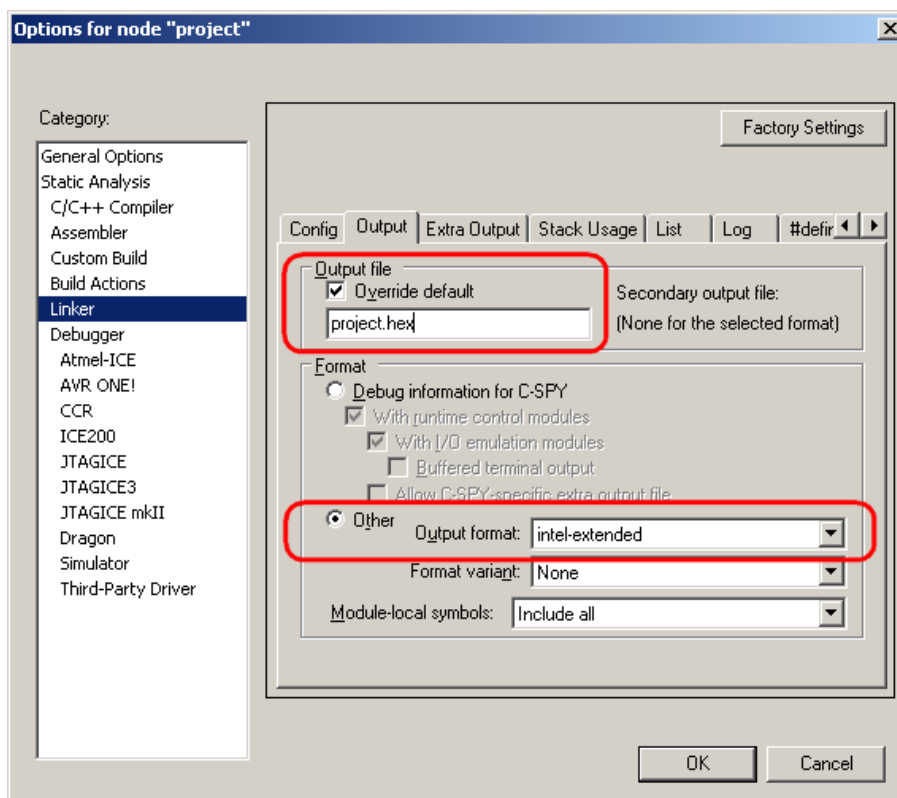
Устанавливаем параметры оптимизации. Выбираем оптимизацию по скорости. Уровень оптимизации устанавливаем «High», для оптимизации кода с директивой inline.



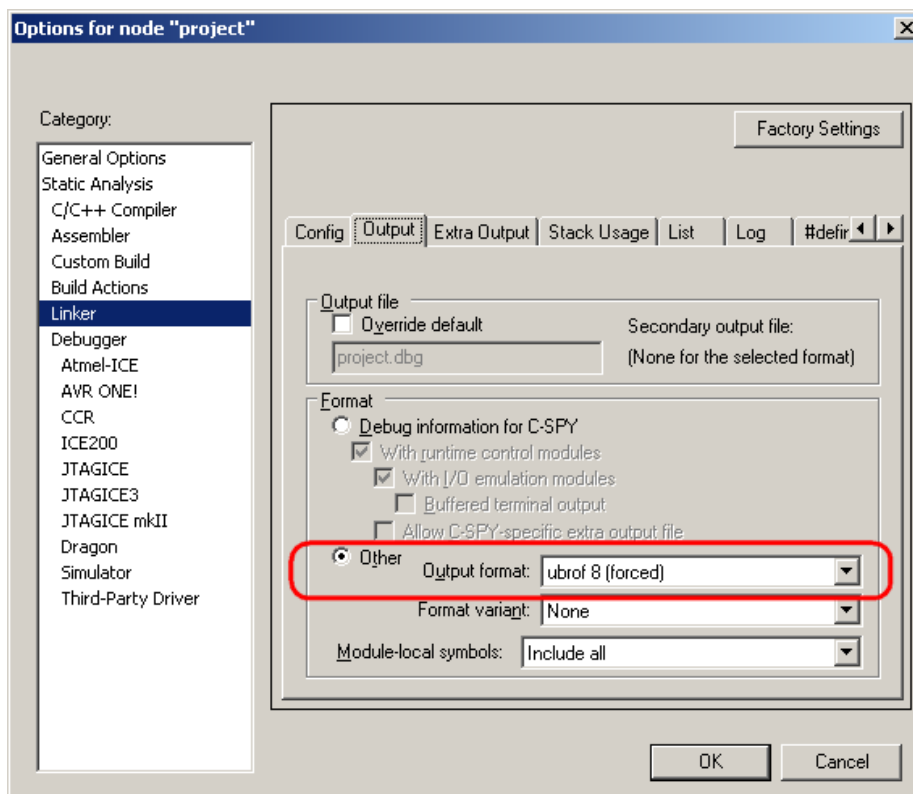
### Резервирование регистров микроконтроллера под собственные нужды.

Регистр R15, для примера, будем использовать в качестве хранилища на 8 флагов. Место хранения флагов будем называть «хранилище флагов», вместо «регистр флагов», чтобы не путать с регистром микроконтроллера.



Настройка выходного файла для конфигурации «Release».Настройка выходного файла для конфигурации «Debug».

Данная настройка параметров позволит в качестве отладчика использовать отладчик AVR Studio 4.18.





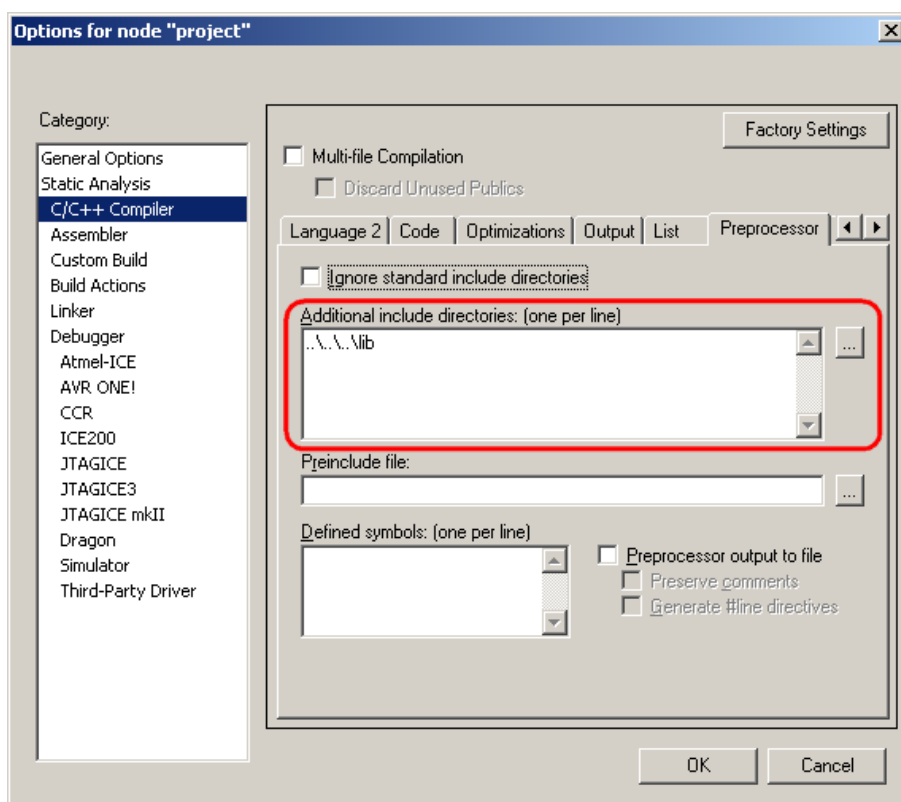
## 2 Принцип построения программ на языке C++

### 2.1 Библиотека MCU

Все классы, не относящиеся непосредственно к алгоритму работы устройства, являются общими для всех проектов и расположены в папке `mcu`. В данной папке классы упорядочены следующим образом:

- все аппаратно независимые классы общего назначения находятся в корне папке `mcu`;
- классы, относящиеся к определённой архитектуре, находятся в отдельных папках (например, `mcu\avr`);
- классы, относящиеся к конкретному микроконтроллеру, находятся в отдельной папке своей архитектуры (например, `mcu\avr\attiny2313`).

Данная библиотека может располагаться как в корневой папке проекта, на одном уровне с файлом `main.cpp`, так и в отдельной папке (для использования в нескольких проектах). Если папка с библиотекой `mcu` располагается не на одном уровне с файлом `main.cpp`, то в настройках IDE IAR Embedded Workbench необходимо прописать путь к библиотеке. При этом корректировка исходных текстов (в части `#include`) не требуется.



### 2.2 Флаги

Одним из значимых элементов любой программы являются флаги. Флагом будем считать один бит информации. Создание хранилища флагов и определение флагов выглядит следующим образом:

```

1 typedef TFlags < 5 > Flags;           // создаём хранилище на 5 флагов
2 typedef TFlag < Flags, 0 > FBell;      // флаг FBell занимает бит 0 в хранилище флагов Flags
3 typedef TFlag < Flags, 1 > FCounter;   // флаг FCounter занимает бит 1 в хранилище флагов Flags
4 typedef TFlagList < Flags, 2, 3 > FLKey; // список флагов FLKey занимает 3 бита начиная со 2 бита

```

Далее флаги могут быть использованы следующим образом:

```

1 if (FCounter::Check())
2 {
3     FBell::Set();
4 }

```

Как видно, обращение к методам происходит без создания экземпляра класса (статически). Отсутствуют конструктор и деструктор.

Для работы с флагами реализованы следующие классы:

TFlags	- класс для создания хранилища флагов в RAM;
TFlagsAVRReg	- класс для создания хранилища на 8 флагов на базе регистра микроконтроллера AVR;
TFlag	- класс для работы с одним флагом;
TFlagList	- класс для работы со списком флагов.

### Класс TFlags

Класс TFlags является аппаратно-независимым и служит для организации хранилища флагов. В качестве хранилища флагов используется массив байт в RAM. Размерность массива определяется на этапе компиляции. Максимальное количество флагов ограничивается объёмом RAM. Скорость работы с одним флагом не зависит от количества флагов в хранилище.

Класс TFlags является шаблонным классом и определяется следующим образом:

```
lib\mcu\TFlags.h
13 template < uint64_t Bits, uint64_t Id = 0 >
14 class TFlags
```

где,

uint64\_t Bits        – количество флагов  
uint64\_t Id = 0     – идентификатор класса

Если с параметром Bits всё понятно, то с параметром Id = 0 могут возникнуть вопросы. Дело в том, что мы используем шаблонные классы, а это значит, что если мы напишем так:

```
1 typedef TFlags < 5 > Flags_In;
2 typedef TFlags < 6 > Flags_Out;
```

то компилятор создаст нам два массива размерностью по одному байту (5 и 6 бит укладываются в один байт). А если мы напишем так:

```
1 typedef TFlags < 5 > Flags_In;
2 typedef TFlags < 5 > Flags_Out;
```

то компилятор создаст нам только один массив размерностью один байт, т.к. входные параметры шаблонного класса одинаковые. Компилятор будет считать, что это один и тот же класс. А что делать, если нам необходимо создать два хранилища с одинаковым количеством флагов? Вот для этого и используется параметр Id, который и будет задавать уникальность определяемого класса. Нужные нам два хранилища флагов мы получим, используя следующий код:

```
1 typedef TFlags < 5, 0 > Flags_In;
2 typedef TFlags < 5, 1 > Flags_Out;
```

В первом случае Id = 0, во втором Id = 1, наши классы стали уникальными.

Некоторые особенности класса TFlags.

```
lib\mcu\TFlags.h
13 template < uint64_t Bits, uint64_t Id = 0 >
14 class TFlags
15 {
16     STATIC_ASSERT(Bits > 0, "TFlags: value is exceed");
17
18     enum { ID = Id };
19
20     typedef STATIC_TYPE_ARRAY_BITS(Bits) T;
21
22     static volatile T Data;
23
24     public:
25         enum { BITS = Bits };
26
27     ...
28 };
29
30 //=====
31 template < uint64_t Bits, uint64_t Id >
32 volatile TFlags< Bits, Id >::T TFlags< Bits, Id >::Data;
```

В строке 16 производится проверка на допустимость количества создаваемых флагов. Первым параметром в макрос `STATIC_ASSERT` передаётся условие, в случае невыполнения которого в окне ошибок компилятора выводится текст ошибки. Текст ошибки указывается во втором параметре макроса. Количество флагов должно быть больше нуля.

В строке 18 фиксируем идентификатор класса.

В строке 20 определяем тип данных `T` для переменной `Data`, которая и является хранилищем флагов (массивом байт). Макросу `STATIC_TYPE_ARRAY_BITS` передаём необходимое нам количество флагов (Bits), а макрос возвращает тип данных (массив байт). Т.е. для 3 флагов будет создан массив, состоящий из одного байта, а для 125 флагов будет создан массив, состоящий из 16 байт.

В строке 22 создаем переменную `Data` с вышеуказанным типом `T`. При определении данной переменной используем ключевое слово `volatile`, т.к. флаги могут изменяться в любом месте программы, в том числе и в прерываниях. Т.к. мы используем статические классы, то все переменные класса необходимо определять отдельно. В строках 84 и 85 определяем переменную `Data`.

В строке 25 фиксируем объём хранилища (максимальное количество флагов для данного хранилища). Эта информация будет использоваться в классе `TFlag` для определения допустимости используемого номера флага (номера бита в массиве байт) для данного хранилища. Т.е. если хранилище рассчитано на 10 флагов, а мы пытаемся определить 11 флагов, то на уровне компиляции будет проведена проверка на допустимость и выдано сообщение об ошибке компиляции.

Из реализованных методов класса `TFlags` для разработчика имеет значение только метод `Init` (установка всех флагов хранилища в требуемое состояние). Остальные методы используются классом `TFlag`.

### Класс `TFlagsAVRReg`

Класс `TFlagsAVRReg` является аппаратно-зависимой разновидностью класса `TFlags`. В качестве хранилища флагов используется один регистр микроконтроллера AVR, освобождённый в настройках компилятора для нужд разработчика. Максимальное количество флагов ограничивается размером регистра, т.е. 8. За счёт использования регистра, а не RAM, скорость работы с флагом становится выше, а код компактнее.

Класс `TFlagsAVRReg` является шаблонным классом и определяется следующим образом:

```
lib\mcu\avr\TFlagsAVRReg.h
10 template < uint8_t Reg >
11 class TFlagsAVRReg
```

где,

`uint8_t Reg` – номер регистра микроконтроллера AVR.

Идентификатор класса `Id` в данном случае не нужен, т.к. номер регистра и так является уникальным.

Некоторые особенности класса `TFlagsAVRReg`.

```
lib\mcu\avr\TFlagsAVRReg.h
13 static volatile __regvar __no_init uint8_t Data @ Reg;
...
73 template < uint8_t Reg >
74 volatile __regvar __no_init uint8_t TFlagsAVRReg < Reg >::Data @ Reg;
```

В строках 13, 73, 74 показано определение регистра микроконтроллера в качестве переменной. Ключевое слово `__no_init` говорит о том, что переменная не будет инициализирована каким либо значением (в переменной мусор). Инициализировать регистр значением можно используя метод `Init`.

Пример использования:

```
1 typedef TFlagsAVRReg < 15 > Flags; // используем регистр R15 для хранения 8 флагов
2 ...
3 Flags::Init(0); // инициализация всех флагов регистра R15 значением 0
```

### Класс `TFlag`

Класс `TFlag` является аппаратно-независимым и предназначен для выполнения каких-либо действий над одним флагом.

Класс TFlag является шаблонным классом и определяется следующим образом:

```
lib\mcu\TFlag.h
13 template < class Flags, uint64_t Flag >
14 struct TFlag
```

где,

class Flags – класс хранилища флагов  
uint64\_t Flag – номер флага в хранилище флагов

Определение одного флага выглядит следующим образом:

```
main.h
1 typedef TFlag < Flags, 0 > FBell; // бит 0 - используется для работы с зуммером
```

где,

FBell – имя флага  
Flags – класс хранилища флагов (на базе классов TFlags или TFlagsAVRReg)  
0 – номер флага в хранилище флагов

Класс реализует следующие методы:

Set – установка флага в лог.1  
Set(bool value) – установка флага в состояние, передаваемое в качестве параметра  
Reset – установка флага в лог.0  
Toggle – инвертирование флага  
Check – проверка флага

Некоторые особенности класса TFlag.

```
lib\mcu\TFlag.h
16 STATIC_ASSERT(Flag < Flags::BITS, "TFlag: amount bit is exceed");
```

В строке 16 производится проверка на допустимость номера флага для данного хранилища. Номер флага не должен превышать объём хранилища.

Если создать два флага с одним и тем же номером, то компилятор не выдаст ошибки, а оба флага будут указывать на один и тот же бит в массиве байт.

Пример использования:

```
main.h
1 typedef TFlagsAVRReg < 15 > Flags;
2 ...
3 typedef TFlag < Flags, 1 > FCounter_1s;
```

```
main.cpp
1 if(Counter_1s::Check())
2 {
3     value16--;
4     Counter_1s::Reset();
5 }
```

## Класс TFlagList

Класс TFlagList является аппаратно-независимым и предназначен для определения списка флагов.

Класс TFlagList является шаблонным классом и определяется следующим образом:

```
lib\mcu\TFlagList.h
12 template < class Flags, uint64_t Start, uint64_t Count >
13 struct TFlagList
```

где,

class Flags – класс хранилища флагов (на базе классов TFlags или TFlagsAVRReg);  
uint64\_t Start – начальный номер флага в хранилище флагов;  
uint64\_t Count – количество флагов.

Пример определения списка флагов:

**main.h**

```
1 typedef TFlagList < Flags, 2, 3 > FLKey;
```

В данном примере для флагов работы с клавиатурой отведены флаги 2,3,4 в хранилище флагов Flags.

Флаги в списке флагов имеют имена F0, F1, F2 и т.д. Доступ к флагам из списка флагов производится следующим образом:

```
1 typedef typename FlagList::F0 Flag_1;
2 typedef typename FlagList::F1 Flag_2;
3
4 Flag_1::Set();
5 Flag_2::Reset();
```

где,

FlagList – список флагов

Flag\_1 – флаг №1

Flag\_2 – флаг №2

При попытке использовать несуществующий номер флага, например F2, компилятор выдаст ошибку, т.к. неиспользуемые флаги с F2 по F49 будут созданы на основе класса TNullType, который не имеет методов.

Некоторые особенности класса TFlagList.

**lib\mcu\TFlagList.h**

```
15 STATIC_ASSERT(((Count > 0) && (Count < 51)), "TFlagList: (Count > 0) && (Count < 51)");
16 STATIC_ASSERT((Start + Count - 1) < Flags::BITS, "TFlagList: amount bit is exceed");
```

В строке 15 производится проверка на допустимость количества задаваемых флагов. Количество задаваемых флагов должно быть в диапазоне от 1 до 50.

В строке 16 производится проверка на допустимость количества занимаемых флагов в хранилище. Т.е. если хранилище флагов рассчитано на 5 флагов, а мы создадим 4 флага начиная со 2, то общее количество флагов составит 6, что превысит объём хранилища.

Пример использования:

**lib\device\DevKey\_Bell.h**

```
18 typedef typename FlagList::F0 FDetect; // флаг "Обнаружено нажатие клавиши"
19 typedef typename FlagList::F1 FHold; // флаг "Удержание"
20 typedef typename FlagList::F2 FPress; // флаг "Клавиша нажата" (для обработки в main)
```

## 2.3 Прерывания

Каждое прерывание оформлено в виде отдельного класса с единственным методом execute. Классы с прерываниями являются аппаратно-зависимыми и описываются в файле IRQ.h. Файл IRQ.h для каждого микроконтроллера свой и располагается в папке с именем микроконтроллера (например, mcu\avr\attiny2313\IRQ.h).

Для примера, класс прерывания IRQ\_Timer0\_CompareA выглядит следующим образом:

**lib\mcu\avr\attiny2313\IRQ.h**

```
46 template < class IRQ >
47 class IRQ_Timer0_CompareA
48 {
49     #pragma vector = 0x1A
50     static __interrupt inline void execute(void)
51     {
52         IRQ::execute();
53     }
54 };
```

Класс IRQ\_Timer0\_CompareA является шаблонным классом. Код, который будет выполняться в прерывании, описывается в отдельном классе и передаётся как параметр шаблона в класс прерывания. Класс с кодом, который будет выполняться в прерывании, реализует единственный метод execute.

Т.к. классы с прерываниями у нас уже описаны в файле `mcu\avr\attiny2313\IRQ.h`, то нам остаётся только создать класс с произвольным именем и передать его в класс прерывания. На практике это выглядит следующим образом:

```

IRQ_1ms.h
1 struct IRQ_1ms
2 {
3     #pragma inline = forced
4     static inline void execute(void)
5     {
6         Counter_1s::Tick();
7         DevBell::Tick();
8         DevLED::Tick();
9         DevKey::Scan();
10    }
11 };
12 ...
13 template void IRQ_Timer0_CompareA < IRQ_1ms > ::execute();

```

В строках 6-9 находится код, который будет выполняться в прерывании. Причём мы не указываем, для какого прерывания предназначен данный код.

В строке 13 мы указываем, что в прерывании `IRQ_Timer0_CompareA` будет выполняться код класса `IRQ_1ms` (метод `execute`). Мы можем передать данный код в любое из прерываний, лишь бы в этом был смысл.

Не стоит думать, что наличие в прерывании нескольких функций, приведёт к увеличению стека. Весь код встраивается с помощью ключевого слова `inline`. Если посмотреть ассемблерный листинг кода прерывания, то можно видеть, что в нём отсутствуют команды `call` (`rcall`).

## 2.4 Драйвера, устройства

Для работы с внешними устройствами, например, клавиатурой, индикатором и т.д., существуют два типа классов:

- аппаратно-зависимый – «драйвер»;
- аппаратно-независимый – «устройство».

Класс «драйвер» обеспечивает доступ к периферии микроконтроллера. Класс «устройство» реализует управление устройством, через методы класса «драйвер», без учёта технических особенностей. Если требуется изменить подключение, например клавиатуры, то достаточно переписать только код класса «драйвер».

Класс «драйвер» передаётся как параметр в шаблонный класс «устройство». Класс «драйвер», при этом, должен обеспечить определённый интерфейс (набор методов), известный классу «устройство».

Класс «драйвер» не имеет шаблонных параметров. Это было сделано сознательно. Всё, что необходимо для работы с устройством на низком уровне, реализуется в самом классе. Параметры настройки классов «драйвер» и «устройство» определяются через `enum` внутри класса. Я предпочитаю самому писать код работы с периферией, используя все возможности низкоуровневого доступа к «железу», нежели доверить эту работу компилятору. Для микроконтроллеров младших семейств универсальность может не подойти как раз по причине раздувания кода.

Пример работы с устройством через драйвер приведён ниже:

#### TDriver.h

```

1 struct TDriver
2 {
3     //-----
4     #pragma inline = forced
5     static inline void Init(void)
6     {
7     ...
8     }
9     //-----
10    #pragma inline = forced
11    static inline void On(void)
12    {
13    ...
14    }
15    //-----
16    #pragma inline = forced
17    static inline void Off(void)
18    {
19    ...
20    }
21 };

```

#### TDevice.h

```

1 template < class Driver >
2 struct TDevice
3 {
4     //-----
5     #pragma inline = forced
6     static inline void Init(void)
7     {
8     ...
9         Driver::Init();
10    ...
11    }
12    //-----
13    #pragma inline = forced
14    static inline void Start(void)
15    {
16    ...
17        Driver::On();
18    ...
19    }
20    //-----
21    #pragma inline = forced
22    static inline void Stop(void)
23    {
24    ...
25        Driver::Off();
26    ...
27    }
28 };

```

#### main.h

```

1 typedef TDevice < TDriver > Dev;

```

#### main.cpp

```

1 Dev::Init();
2 Dev::Start();
3 Dev::Stop();

```

В классе TDriver реализованы методы Init, On, Off, из которых непосредственно производится доступ к периферии микроконтроллера. Данные методы определены как public (используется ключевое слово struct), т.е. будут доступны для класса TDevice.

Класс TDevice реализует аппаратно-независимое управление устройством, используя методы класса TDriver. Класс TDriver передаётся в качестве параметра шаблонному классу TDevice.

В файле `main.h` производится связывание классов «драйвер» и «устройство». Результатом данного связывания является класс `Dev`, включающий в себя весь функционал работы с устройством.

В файле `main.cpp` показана работа с устройством. В строке 1 происходит инициализация устройства. В строках 2 и 3 производится вызов методов `Start` и `Stop` данного устройства.

## 2.5 Периферия микроконтроллеров AVR

Работа с периферией микроконтроллера также происходит через классы. Рассмотрим, как можно организовать работу с периферией микроконтроллера на примере сторожевого таймера (WDT) и таймера (Timer0).

### 2.5.1 Сторожевой таймер (WDT)

Работу со сторожевым таймером описывает класс `TWDT`. Данный класс является аппаратно-зависимым и описывается в файле `TWDT.h`. Файл `TWDT.h` для каждого микроконтроллера свой и располагается в папке с именем микроконтроллера (например, `mcu\avr\attiny2313\TWDT.h`).

Класс реализует следующие методы:

```
void Enable(const TTimeout &value = Timeout_8000ms)
```

#### Описание

Включение сторожевого таймера с заданным значением срабатывания. По умолчанию таймаут срабатывания установлен на 8 секунд.

Для микроконтроллера `ATtiny2313` доступны следующие значения:

```
TWDT::Timeout_16ms
TWDT::Timeout_32ms
TWDT::Timeout_64ms
TWDT::Timeout_125ms
TWDT::Timeout_250ms
TWDT::Timeout_500ms
TWDT::Timeout_1000ms
TWDT::Timeout_2000ms
TWDT::Timeout_4000ms
TWDT::Timeout_8000ms
```

#### Пример использования

```
TWDT::Enable();
TWDT::Enable(TWDT::Timeout_500ms);
```

#### Расположение

```
mcu\avr\attiny2313\TWDT.h
```

```
void Disable(void)
```

#### Описание

Выключение сторожевого таймера.

#### Пример использования

```
TWDT::Disable();
```

#### Расположение

```
mcu\avr\attiny2313\TWDT.h
```

```
void Change(const TTimeout &value)
```

#### Описание

Изменение значения таймаута срабатывания сторожевого таймера.

#### Пример использования

```
TWDT::Change(TWDT::Timeout_500ms);
```

#### Расположение

```
mcu\avr\attiny2313\TWDT.h
```



```
void Reset(void)
```

#### Описание

Сброс сторожевого таймера. Выполнение данного метода равносильно выполнению ассемблерной команды WDR.

#### Пример использования

```
TWDT::Reset();
```

#### Расположение

```
mcu\avr\attiny2313\TWDT.h
```

Пример использования:

```
main.cpp
```

```
1 void main(void)
2 {
3     ...
4     TWDT::Enable();
5     _SEI();
6
7     while(1)
8     {
9         TWDT::Reset();
10    ...
11    }
12 }
```

### 2.5.2 Таймеры (Timer0)

Работу с таймером Timer0 описывает класс TTimer0. Данный класс является аппаратно-зависимым и описывается в файле TTimer0.h. Файл TTimer0.h для каждого микроконтроллера свой и располагается в папке с именем микроконтроллера (например, mcu\avr\attiny2313\TTimer0.h).

Для работы с таймером реализованы дополнительные классы:

```
struct TTimerConst
```

#### Описание

Класс определяет константы TimerValueHz и TimerValueSec для идентификации типа единиц измерения (частота или время).

#### Расположение

```
mcu\avr\TTimerConst.h
```

```
template < uint64_t value >
struct TTimerValue : private TTimerConst
```

#### Описание

Класс определяет константы ns, us, ms, Hz, KHz, MHz, Cycles, KCycles, MCycles для задания времени срабатывания таймера.

#### Пример использования

```
typedef TTimer0_SimpleCycle < TTimerValue < 1 > ::ms > Timer_1ms;
```

#### Расположение

```
mcu\avr\TTimerConst.h
```

```
template < uint64_t value >
struct TTimerCycles : private TTimerConst
```

#### Описание

Класс осуществляет перевод значений из единиц измерения времени, в циклы микроконтроллера.

#### Пример использования

```
enum { CYCLES = TTimerCycles < value > ::Cycles };
```

#### Расположение

```
mcu\avr\TTimerConst.h
```

```
template < uint64_t value >
```

```
struct TTimer0_CalcReg : TTimer0
```

### Описание

Класс реализует расчёт значений регистров, определяющих время срабатывания таймера для различных режимов работы.

Если срабатывание с заданным значением времени не может быть реализовано, при текущих режимах работы микроконтроллера (кварц, fuse bits), то на этапе компиляции будет выдана ошибка.

### Пример использования

```
enum { OCR = TTimer0_CalcReg < value > ::OCR };
enum { DIV = TTimer0_CalcReg < value > ::DIV };
```

### Расположение

```
mcu\avr\attiny2313\TTimer0.h
```

Для корректной работы таймера необходимо в файле system.h установить частоту, на которой работает наш микроконтроллер.

### system.h

```
1 #define F_CLK 8000000ULL
```

### **Класс TTimer0\_SimpleCycle.**

На базе класса TTimer0 создан класс TTimer0\_SimpleCycle, реализующий простой циклический таймер.

### lib\mcu\avr\attiny2313\TTimer0\_SimpleCycle.h

```
18 template < uint64_t value >
19 class TTimer0_SimpleCycle : private TTimer0
20 {
21     enum { OCR = TTimer0_CalcReg < value > ::OCR };
22     enum { DIV = TTimer0_CalcReg < value > ::DIV };
23     public:
24     ...
27     #pragma inline = forced
28     static inline void Init(void)
29     {
30         TTimer0::SetConfig(Mode_CTC);
31         TTimer0::SetCounterValue(0);
32         TTimer0::SetCompareOCRValue(OCR);
33         TTimer0::SetInterruptCompareAFlag(false);
34         TTimer0::SetInterruptCompareA(true);
35     }
36     ...
39     #pragma inline = forced
40     static inline void Start(void)
41     {
42         TTimer0::SetCounterValue(0);
43         TTimer0::SetClockDivider(static_cast < TDivider > (DIV));
44     }
45     ...
53 }
```

Используя методы класса TTimer0, реализуем нужный функционал.

В строке 18 значение value определяет время срабатывания таймера (параметр шаблонного класса).

В строках 21 и 22 рассчитываем значение регистра OCR и делителя.

В строке 30 устанавливаем режим работы таймера (CTC).

В строке 31 устанавливаем значение регистра счётчика TCNT0 в 0.

В строке 32 записываем рассчитанное значение OCR в регистр OCR0A.

В строке 33 сбрасываем флаг прерывания OCF0A в регистре TIFR.

В строке 34 разрешаем прерывания OCIE0A в регистре TIMSK.

В строке 42 устанавливаем значение регистра счётчика TCNT0 в 0.

В строке 43 записываем рассчитанное значение делителя в регистр TCCR0B.

### Пример использования:

**main.h**

```
1 | typedef TTimer0_SimpleCycle < TTimerValue < 1 > ::ms > Timer_1ms;
```

**main.cpp**

```
1 | void main(void)
2 | {
3 |   ...
4 |   Timer_1ms::Init();
5 |   ...
6 |   Timer_1ms::Start();
7 |   ...
8 |   _SEI();
9 |
10 |   while(1)
11 |   {
12 |     ...
13 |   }
14 | }
```

В файле main.h производим настройку простого цикличного таймера, построенного на базе таймера 0, на прерывание каждую 1 ms.

В файле main.cpp в строке 4 производим инициализацию цикличного таймера, а в строке 6 запуск.

Т.к. класс TTimer0\_SimpleCycle использует режим работы таймера CTC, то каждую 1 ms будет срабатывать прерывание IRQ\_Timer0\_CompareA (0x1A). Работа с прерываниями описана в п.2.3.

### 2.6 Результаты компиляции

Сравнение результатов компиляции исходных кодов на языках Си и C++, показал, что бинарный код, при оптимизации по размеру (size) для языка Си, как правило, получается меньше, чем для языка C++. Причиной этого может быть как использование ключевого слова inline, так и особенности компилятора C++.

### 3 Пример программы на языке C++ (динамическая индикация)

#### 3.1 Описание

Рассмотрим пример построения программы на языке C++, демонстрирующий управление клавиатурой, зуммером и 7-сегментным индикатором. Схемы проверки работоспособности кода представлены на рисунках 3.1 и 3.2. Для упрощения, в схемах отсутствуют элементы, необходимые для работы в реальном устройстве.

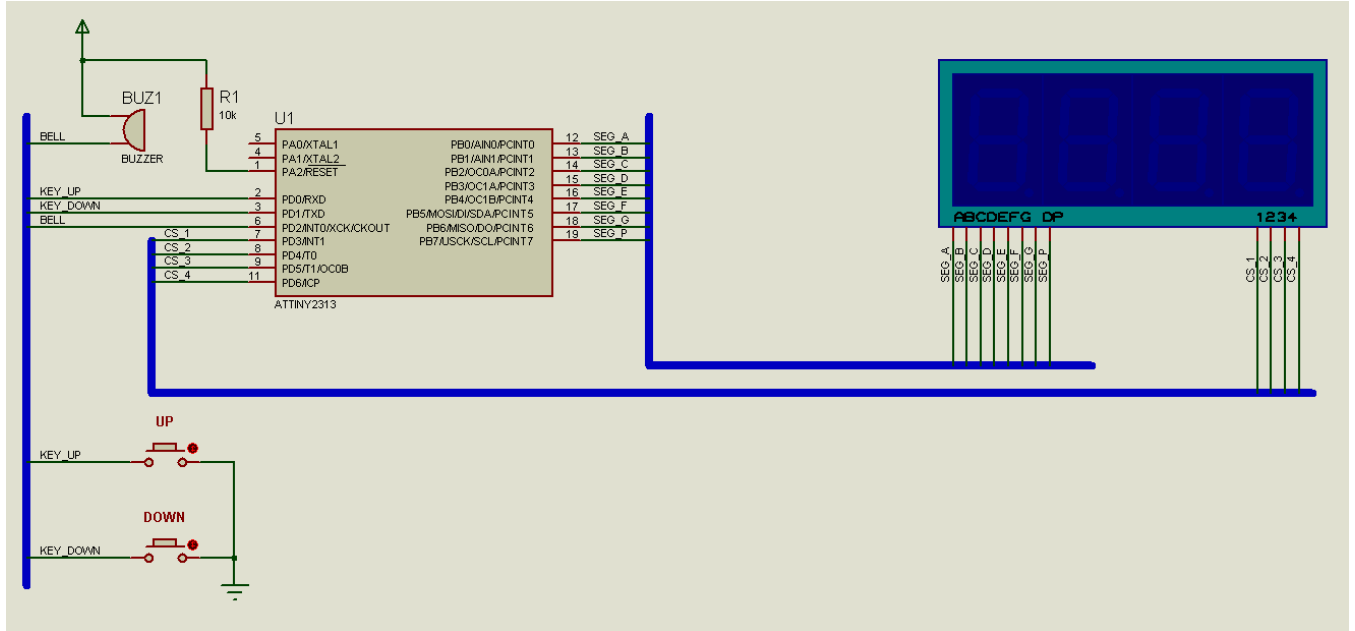


Рисунок 3.1 Подключение LED-индикатора к микроконтроллеру ATtiny2313

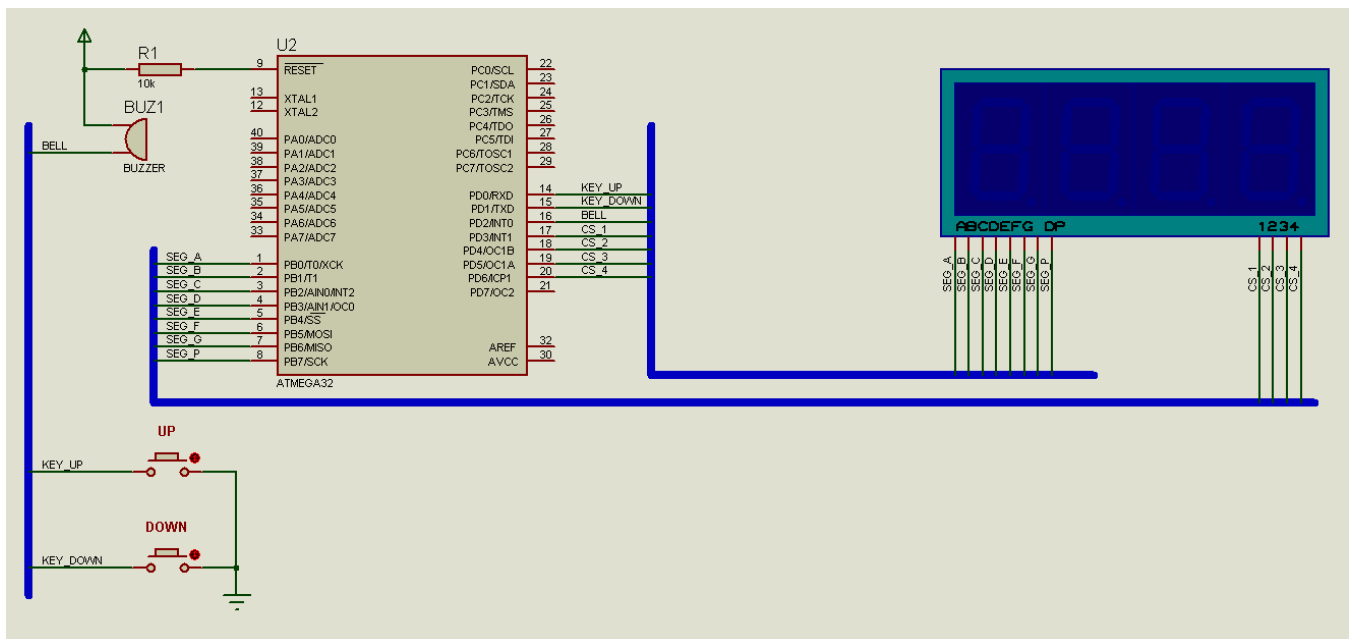


Рисунок 3.2 Подключение LED-индикатора к микроконтроллеру ATmega32

Для расчёта временных интервалов необходимо задать частоту, на которой работает наш микроконтроллер. Данный параметр задаётся в файле `system.h`.

`system.h`

```
7 | #define F_CLK 8000000ULL
```

Данный параметр используется для расчёта задержек в макросах `DELAY_NS`, `DELAY_US`, `DELAY_MS`, `DELAY_S`, `DELAY_TICK`, а также для расчёта значений регистров настройки таймеров.

В программе реализован циклический таймер на 1 ms. В прерывании данного таймера выполняется:

- отсчёт временных интервалов для формирования события каждую секунду;
- отсчёт временных интервалов для формирования длительности звучания зуммера;
- опрос клавиатуры с формированием кода нажатой клавиши;
- вывод данных на 7-сегментный индикатор.

### 3.2 Настройка периферии микроконтроллера

Настройка периферии микроконтроллера производится в файле main.h.

main.h

```

7 #include "system.h"
8
9 #include "mcu/mcu.h"
10 #include "mcu/TFlag.h"
11 #include "mcu/TFlagList.h"
12 #include "mcu/TFlags.h"
13 #include "mcu/avr/TFlagsAVRReg.h"
14 #include "mcu/TCounter.h"
15
16 #include "driver/TDrvBell.h"
17 #include "driver/TDrvKey.h"
18 #include "driver/TDrvLED_7Seg.h"
19
20 #include "device/TDevBell.h"
21 #include "device/TDevKey_Bell.h"
22 #include "device/TDevLED_7Seg.h"
23
24 //----- Флаги -----
25 //typedef TFlags < 5 > Flags;
26 typedef TFlagsAVRReg < 15 > Flags;
27 typedef TFlag < Flags, 0 > FBell;
28 typedef TFlag < Flags, 1 > FCounter_1s;
29 typedef TFlagList < Flags, 2, 3 > FLKey;
30 //----- Периферия -----
31 typedef TDevBell < TDrvBell, FBell > DevBell;
32 typedef TDevKey_Bell < TDrvKey, FLKey, DevBell > DevKey;
33 typedef TDevLED_7Seg < TDrvLED_7Seg > DevLED;
34 //----- Прочее -----
35 typedef TCounter < 1000, FCounter_1s > Counter_1s;
36 //-----
37 typedef TTimer0_SimpleCycle < TTimerValue < 1 > ::ms > Timer_1ms;
38
39 #include "IRQ_1ms.h"

```

В строке 7 подключаем заголовочный файл system.h с системными параметрами (в нашем случае, это только частота работы микроконтроллера F\_CLK).

#### Подключаем заголовочные файлы из библиотеки mcu.

В строке 9 подключаем заголовочный файл mcu.h из библиотеки mcu (для подключения необходимых заголовочных файлов, соответствующих нашему микроконтроллеру). После выбора микроконтроллера в IDE, при компиляции подключаются необходимые заголовочные файлы (ну если, конечно, они у вас есть). Таким образом, при смене типа микроконтроллера, нет необходимости в корректировке заголовочных файлов или прописывании путей.

В строке 10 подключаем заголовочный файл TFlag.h для работы с отдельными флагами.

В строке 11 подключаем заголовочный файл TFlagList.h для работы со списком флагов.

В строке 12 подключаем заголовочный файл TFlags.h для работы с хранилищем флагов, размещённым в области RAM (класс TFlags в примере не используется – закомментирован).

В строке 13 подключаем заголовочный файл TFlagsAVRReg.h для работы с хранилищем флагов, размещённым в регистре микроконтроллера.

В строке 14 подключаем заголовочный файл TCounter.h для работы с счётчиком.

#### Подключаем заголовочные файлы драйверов.

В строке 16 подключаем заголовочный файл TDrvBell.h драйвера зуммера.

В строке 17 подключаем заголовочный файл TDrvKey.h драйвера клавиатуры.

В строке 18 подключаем заголовочный файл TDrvLED\_7Seg.h драйвера 7-сегментного индикатора.

Подключаем заголовочные файлы устройств.

В строке 20 подключаем заголовочный файл TDevBell.h устройства зуммера.

В строке 21 подключаем заголовочный файл TDevKey\_Bell.h устройства клавиатуры с поддержкой звукового сигнала при нажатии кнопки.

В строке 22 подключаем заголовочный файл TDevLED\_7Seg.h устройства 7-сегментного индикатора.

Определяем флаги, необходимые для работы нашего примера.

В строке 25 создаём хранилище на 5 флагов в RAM (закомментирована).

В строке 26 создаём хранилище на 8 флагов в регистре R15 микроконтроллера.

В строке 27 определяем флаг FBell. Флаг будет использоваться в устройстве «Зуммер». Флаг будет взведён на время звучания звукового сигнала.

В строке 28 определяем флаг FCounter\_1s. Флаг будет использоваться в устройстве «Счётчик». Данный флаг будет взводиться 1 раз в секунду для формирования события, а сбрасывать после обработки события.

В строке 29 определяем список флагов FLKey. Данный список флагов будет использоваться в устройстве «Клавиатура».

Конечно, флаги можно было определять непосредственно в классе устройства, но тогда увеличится расход RAM, т.к. пришлось бы выделить под флаги дополнительно по одному байту для каждого устройства.

Определяем устройства.

В строке 31 определяем класс DevBell, для работы с зуммером.

В строке 32 определяем класс DevKey, для работы с клавиатурой.

В строке 33 определяем класс DevLED, для работы с 7-сегментным индикатором.

Настройка счётчика.

В строке 35 определяем класс Counter\_1s. В качестве параметров шаблонного класса передаём размерность счётчика и флаг.

Настройка таймера на 1 миллисекунду.

В строке 37 определяем класс Timer\_1ms. В качестве параметра шаблонного класса передаём значение периодичности срабатывания циклического таймера.

Классы прерываний.

В строке 39 подключаем заголовочный файл IRQ\_1ms.h с кодом класса, который будет выполняться в прерывании каждую 1 ms. Данная строка подключается последней, чтобы все устройства, код которых будет использоваться в прерывании, были уже определены.

### 3.3 Счётчик

Для формирования события каждую секунду используем счётчик (класс TCounter).

Определение счётчика выглядит следующим образом:

```
main.h
25 typedef TFlags < 5 > Flags;
...
28 typedef TFlag < Flags, 1 > FCounter_1s;
...
35 typedef TCounter < 1000, FCounter_1s > Counter_1s;
```

Т.к. увеличение счётчика будет происходить в прерывании каждую миллисекунду, то для получения счётчика на 1 s (1000 ms) необходимо создать счётчик на 1000 единиц.

Для работы счётчика необходимо создать флаг (FCounter\_1s), который будет взводиться каждую секунду. Сброс флага происходит после обработки события.

Пример использования приведён ниже:

**main.cpp**

```
27 if(Counter_1s::Check())
28 {
    ...
30     Counter_1s::Reset();
31 }
```

В строке 27 проверяем флаг наличия события (прошла 1 секунда или нет).

В строке 30 сбрасываем флаг события.

Увеличение счётчика происходит в прерывании каждую 1 ms.

**IRQ\_1ms.h**

```
17 Counter_1s::Tick();
```

### 3.4 Зуммер

В примере используется пьезоизлучатель. Т.е. формирование звукового сигнала происходит при подаче постоянного напряжения на зуммер. В примере зуммер используется только для формирования звукового сигнала при нажатии на кнопку.

Т.к. работа с устройствами у нас производится через классы драйвера и устройства, то необходимо создать данные классы.

Класс драйвера TDrvBell поддерживает следующие методы:

Init	- настройка портов ввода/вывода
On	- включение звукового сигнала
Off	- выключение звукового сигнала
Toggle	- изменение состояния звукового сигнала

Класс устройства TDevBell поддерживает следующие методы:

Start	- включение зуммера на время t (в миллисекундах)
Tick	- отсчёт времени включенного зуммера и его выключение (выполняется в прерывании 1 ms)

Настройка зуммера приведена ниже:

**main.h**

```
25 typedef TFlags < 5 > Flags;
    ...
27 typedef TFlag < Flags, 0 > FBell;
    ...
31 typedef TDevBell < TDrvBell, FBell > DevBell;
```

Отсчёт временных интервалов производится в прерывании каждую 1 ms.

**IRQ\_1ms.h**

```
18 DevBell::Tick();
```

### 3.5 Клавиатура

В примере клавиатура состоит из двух кнопок «UP» и «DOWN». С помощью данных кнопок будем менять значение переменной value16, которая в дальнейшем будет выведена на индикатор. Работа с клавиатурой реализована в классах TDrvKey и TDevKey\_Bell.

Класс драйвера TDrvKey поддерживает следующие методы:

Init	- инициализация портов ввода/вывода
GetRawCode	- чтение кода нажатой клавиши без дополнительной обработки. В случае нажатия клавиши возвращается ее код, в случае отсутствия нажатия клавиши, либо нажатия двух и более клавиш, возвращается код 0xFF.

Коды клавиш определены через enum:

**driver\TDrvKey.h**

```
24 enum
25 {
26     KEY_UP    = 0,
27     KEY_DOWN  = 1,
28     KEY_OFF   = 0xFF
29 };
```

Класс устройства TDevKey\_Bell поддерживает следующие методы:

- Check - проверка флага нажатой клавиши с учётом обработки (дребезг, автоповтор)
- Reset - сброс флага нажатой клавиши (выполняется после обработки события нажатой клавиши)
- GetCode - чтение кода нажатой клавиши с учётом обработки
- Scan - сканирование клавиатуры (выполняется в прерывании 1 ms, для обеспечения временных интервалов)

Класс TDevKey\_Bell позволяет установить некоторые константы работы с клавиатурой:

lib\device\TDevKey\_Bell.h

```
13 enum { DELAY_1PRESS = 700ULL }; // задержка после первого нажатия, мсек
14 enum { DELAY_HOLD = 100ULL }; // задержка при автоповторе, мсек
15 enum { DELAY_NOISE = 20ULL }; // подавление дребезга контактов, мсек
16 enum { DURATION_BELL = 1ULL }; // длительность звучания зуммера, мсек
```

Настройка клавиатуры приведена ниже:

main.h

```
25 typedef TFlags < 5 > Flags;
...
29 typedef TFlagList < Flags, 2, 3 > FLKey;
...
32 typedef TDevKey_Bell < TDrvKey, FLKey, DevBell > DevKey;
```

В строке 29 определяем список флагов, необходимых для работы клавиатуры.

В строке 32 определяем класс DevKey для работы с клавиатурой. В шаблонный класс TDevKey\_Bell также передаём класс устройства DevBell для формирования звукового сигнала при нажатии кнопок.

Опрос кнопок производится в прерывании каждую 1 ms.

IRQ\_1ms.h

```
20 DevKey::Scan();
```

Пример использования приведён ниже:

main.cpp

```
33 if (DevKey::Check())
34 {
35     uint8_t key = DevKey::GetCode();
36     switch (key)
37     {
38         case DevKey::KEY_UP : value16++; break;
39         case DevKey::KEY_DOWN : value16--; break;
40     }
41     DevKey::Reset();
42 }
```

### 3.6 7-сегментный индикатор

7-сегментный индикатор состоит из 4 разрядов. Работа с индикатором реализована в классах TDrvLED\_7Seg и TDevLED\_7Seg.

Класс драйвера TDrvLED\_7Seg поддерживает следующие методы:

- Init - инициализация портов ввода/вывода
- Tick - вывод одного символа на индикатор (выполняется в прерывании 1 ms)
- GetLEDDigit - конвертация цифр в код 7-сегментного индикатора
- GetLEDSymbol - конвертация символов в код 7-сегментного индикатора

Коды символов описываются через enum:

driver\TDrvLED\_7Seg.h

```
60 enum { SYM_SPACE = 0, SYM_MINUS = 1 };
```

Класс устройства TDevLED\_7Seg поддерживает следующие методы:

- WriteNum - запись числа в буфер дисплея
- WriteDigit - запись цифры в буфер дисплея
- WriteSymbol - запись символа в буфер дисплея

При записи данных в буфер дисплея код преобразуется в код 7-сегментного индикатора (для сокращения выполняемого кода в прерывании).



Пример настройки и использования приведён ниже:

**main.h**

```
33 typedef TDevLED_7Seg < TDrvLED_7Seg > DevLED;
```

**main.cpp**

```
10 void main(void)
11 {
12     uint16_t value16 = 9999;
13     ...
16     DevLED::Init();
17     ...
22     _SEI();
23
24     while(1)
25     {
26         ...
43         DevLED::WriteNum(value16);
44     }
45 }
```

### 3.7 Прерывания

В примере используется прерывание таймера 0 в режиме CTC. На данном таймере реализуется прерывание каждую 1ms.

Настройка таймера 0 на 1 ms приведён ниже:

**main.h**

```
37 typedef TTimer0_SimpleCycle < TTimerValue < 1 > ::ms > Timer_1ms;
```

Код, выполняемый в прерывании таймера 0, приведён ниже:

**IRQ\_1ms.cpp**

```
12 struct IRQ_1ms
13 {
14     #pragma inline = forced
15     static inline void execute(void)
16     {
17         Counter_1s::Tick();
18         DevBell::Tick();
19         DevLED::Tick();
20         DevKey::Scan();
21     }
22 };
```

Привязка данного кода к конкретному прерыванию выполняется следующим образом:

**IRQ\_1ms.cpp**

```
25 template void IRQ_Timer0_CompareA < IRQ_1ms > ::execute();
```

### 3.8 Основной цикл программы

Код файла main.cpp приведён ниже:

main.cpp

```
5  #include "main.h"
   ...
10 void main(void)
11 {
12     uint16_t value16 = 9999;
13
14     DevBell::Init();
15     DevKey::Init();
16     DevLED::Init();
17     Timer_1ms::Init();
18     Counter_1s::Init();
19
20     Timer_1ms::Start();
21     TWDT::Enable();
22     _SEI();
23
24     while(1)
25     {
26         TWDT::Reset();
27         if(Counter_1s::Check())
28         {
29             value16--;
30             Counter_1s::Reset();
31         }
32
33         if(DevKey::Check())
34         {
35             uint8_t key = DevKey::GetCode();
36             switch(key)
37             {
38                 case DevKey::KEY_UP : value16++; break;
39                 case DevKey::KEY_DOWN : value16--; break;
40             }
41             DevKey::Reset();
42         }
43         DevLED::WriteNum(value16);
44     }
45 }
```

В строке 5 подключаем заголовочный файл main.h с описанием оборудования.

В строке 12 определяем переменную value16, для последующей корректировки и отображения.

В строке 14 инициализируем устройство «Зуммер».

В строке 15 инициализируем устройство «Клавиатура».

В строке 16 инициализируем устройство «7-сегментный индикатор».

В строке 17 инициализируем таймер на 1 ms.

В строке 18 инициализируем счётчик на 1 s.

В строке 20 запускаем таймер на 1 ms.

В строке 21 включаем сторожевой таймер.

В строке 22 разрешаем прерывания.

В строке 26 сбрасываем сторожевой таймер.

В строке 27 проверяем флаг формирования события «1 секунда».

В строке 29 уменьшаем значение переменной value16.

В строке 30 сбрасываем флаг формирования события «1 секунда». Событие обработано.

В строке 33 проверяем флаг нажатия клавиш.

В строке 35 считываем код нажатой клавиши.

В строке 38 выполняем действия при нажатии на клавишу «UP».

В строке 39 выполняем действия при нажатии на клавишу «DOWN».

В строке 41 сброс флага нажатия клавиш. Обработка нажатой клавиши выполнена.

В строке 43 выводим на индикатор значение переменной value16.

### 3.9 Результаты компиляции проекта

Для эмуляции в Proteus для данного проекта создано две схемы:

- ATtiny2313;
- ATmega32.

Т.к. для подключения периферии к микроконтроллерам используются одинаковые имена портов ввода/вывода, то корректировка исходных текстов не требуется (за исключением имени вектора прерывания таймера 0). Выбор микроконтроллера производится в настройках проекта согласно п.1.3.

Результат компиляции проекта на ATtiny2313:

Memory	Модель памяти Tiny	
	Оптимизация High + Speed	Оптимизация High + Size
Flash, байт	737	747
RAM, байт	78	

Результат компиляции проекта на ATmega32:

Memory	Модель памяти Tiny		Модель памяти Small	
	Оптимизация High + Speed	Оптимизация High + Size	Оптимизация High + Speed	Оптимизация High + Size
Flash, байт	795	807	797	839
RAM, байт	78			

## 4 Пример программы на языке C++ (символьный LCD-индикатор на контроллере HD44780)

### 4.1 Описание

Рассмотрим пример построения программы на языке C++, демонстрирующий работу с LCD-индикатором на контроллере HD44780. Работу с клавиатурой и зуммером мы рассмотрели ранее и повторяться не будем. Схемы проверки работоспособности кода представлены на рисунках 4.1 – 4.3. Для упрощения, в схемах отсутствуют элементы, необходимые для работы в реальном устройстве.

Рассмотрим следующие моменты работы с LCD-индикатором:

- 4-проводная схема подключения;
- 8-проводная схема подключения;
- подключение LCD-индикатора с двумя контроллерами HD44780 (40 символов 4 строки);
- конвертирование текста для LCD-индикатора.

Мы не будем использовать LCD-индикатор в режиме чтения (вход R/W подключен к GND).

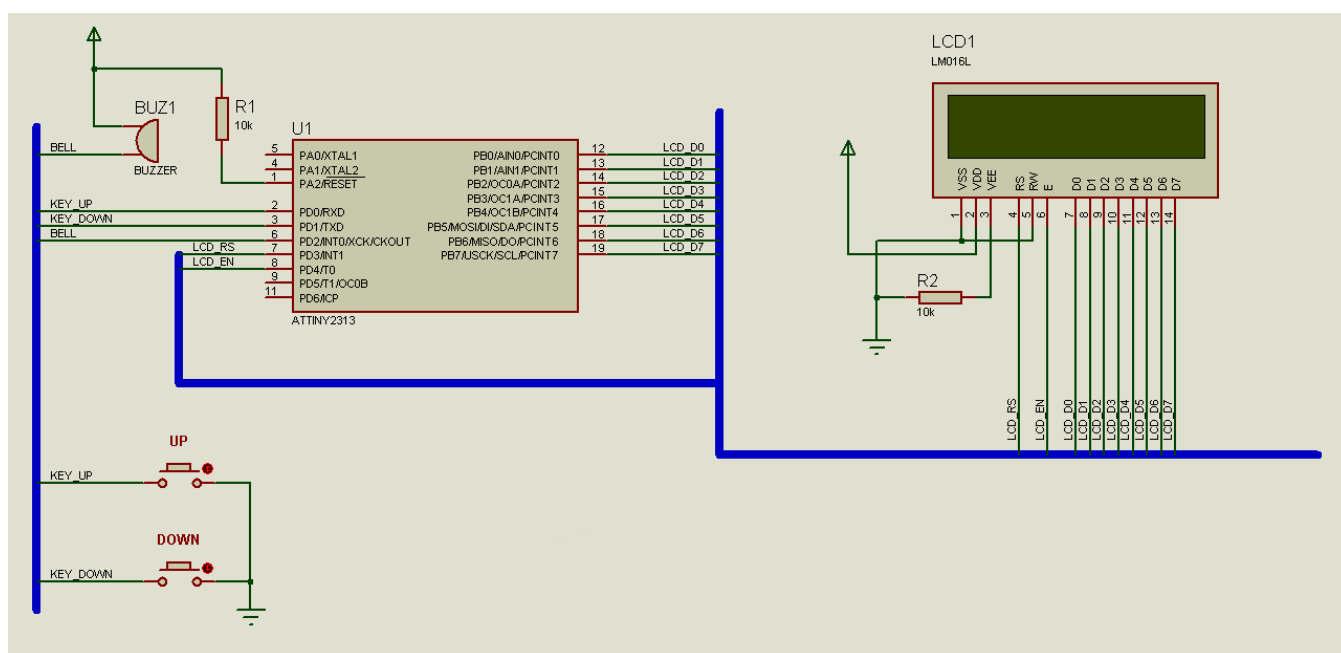


Рисунок 4.1 Подключение индикатора LCD1602 к микроконтроллеру ATtiny2313

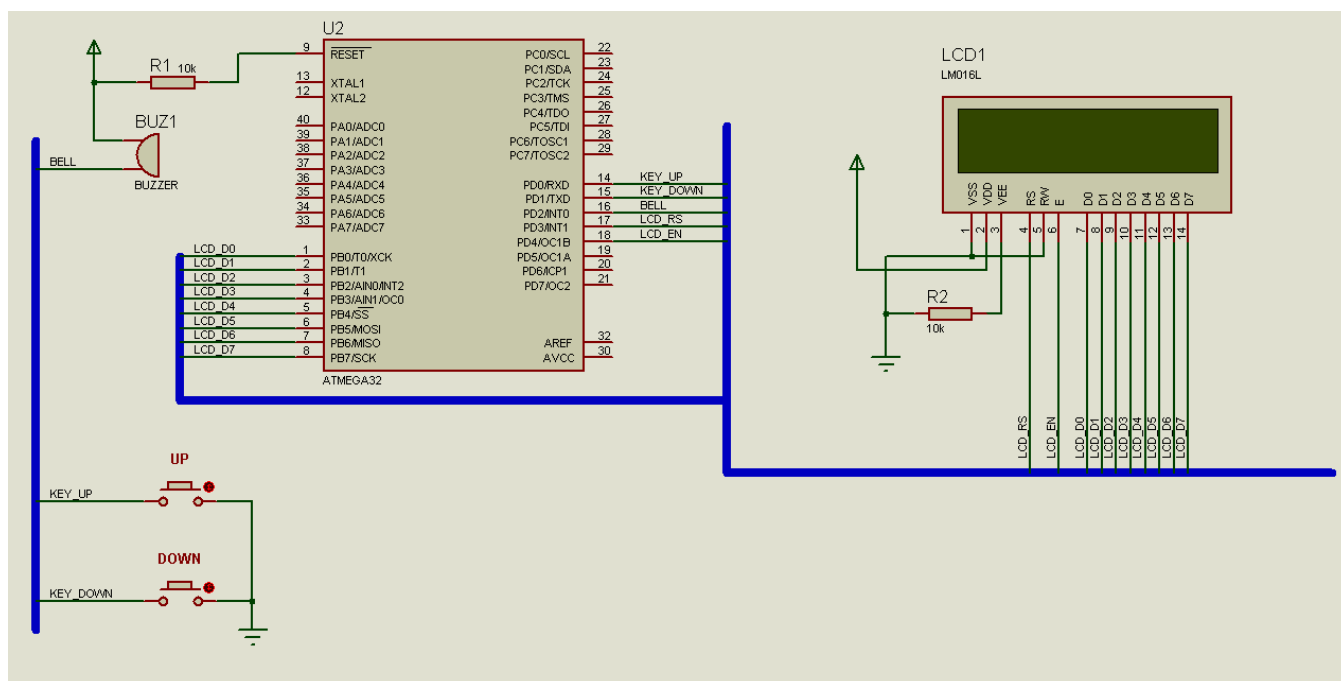


Рисунок 4.2 Подключение индикатора LCD1602 к микроконтроллеру ATmega32

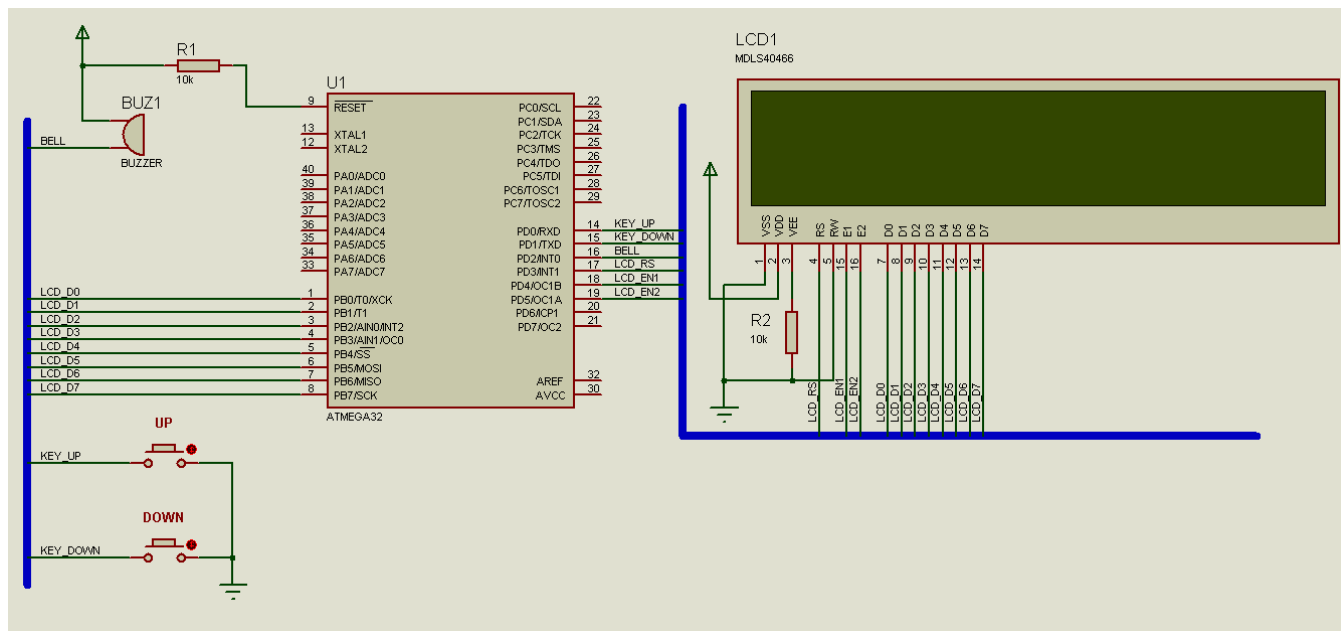


Рисунок 4.3 Подключение индикатора LCD4004 к микроконтроллеру ATmega32

### Организация вывода данных на LCD-индикатор.

Для работы с индикатором создаётся однобайтный массив (буфер) в RAM, соответствующий количеству символов индикатора. Все функции вывода текста на индикатор работают только с буфером. Непосредственно вывод данных на индикатор будет производиться в прерывании каждую 1 ms. За одно прерывание в индикатор записывается один байт данных.

Такая организация работы с индикатором позволит нам:

- избежать задержек при работе с индикатором (согласно datasheet на LCD-индикатор задержка после записи одного байта составляет ~40 us);
- подключить к выводам микроконтроллера дополнительную периферию (на шину D0-D7 индикатора).

## 4.2 Драйвер «LCD-индикатор»

Для работы с индикатором необходимо создать драйвер, реализующий определённый интерфейс. Драйвер должен поддерживать следующие публичные методы:

Init	- инициализация портов ввода/вывода
Write	- запись байта в индикатор
SetModeCmd	- установка режима записи команд
SetModeData	- установка режима записи данных

У нас уже есть следующие классы драйверов:

TDrvLCD_HD44780_4bit	- 4-проводная шина данных, один контроллер HD44780
TDrvLCD_HD44780_8bit	- 8-проводная шина данных, один контроллер HD44780
TDrvLCD_HD44780x2_4bit	- 4-проводная шина данных, два контроллера HD44780
TDrvLCD_HD44780x2_8bit	- 8-проводная шина данных, два контроллера HD44780

Подробно описывать код данных драйверов наверно не нужно (всё вроде понятно). Хочу отметить только один момент: в каждом драйвере есть признак количества используемых в индикаторе контроллеров HD44780.

```
driver\TDrvLCD_HD44780_8bit.h
```

```
19 enum { HD44780 = 1 }; // количество контроллеров HD44780 в индикаторе
```

```
driver\TDrvLCD_HD44780x2_8bit.h
```

```
19 enum { HD44780 = 2 }; // количество контроллеров HD44780 в индикаторе
```

Проверка данного признака не позволит подключить драйвер для работы с одним контроллером к индикатору с двумя контроллерами. Например, следующий код приведёт к ошибке компиляции:

```
1 typedef TDevLCD_HD44780_4004 < TDrvLCD_HD44780_8bit > DevLCD;
```

Правильно будет так:

```
1 | typedef TDevLCD_HD44780_4004 < TDrvLCD_HD44780x2_8bit > DevLCD;
```

### 4.3 Устройство «LCD-индикатор»

Для работы с устройством «LCD-индикатор» были созданы следующие классы:

TDevLCD_HD44780_Base	- базовый класс для работы с индикатором
TDevLCD_HD44780x1	- класс для работы с индикаторами построенных на одном контроллере HD44780
TDevLCD_HD44780x2	- класс для работы с индикаторами построенных на двух контроллерах HD44780
TDevLCD_HD44780_TypeXXXX	- серия классов, определяющих параметры конкретной модели индикатора
TDevLCD_HD44780_XXXX	- серия классов, упрощающая настройку индикатора

#### Класс TDevLCD\_HD44780\_Base

Данный класс определяет общие для всех индикаторов методы. Реализованы следующие методы:

Init	- инициализация индикатора
SetPos	- установка начальной позиции вывода текста (работа через буфер)
LoadSymbol	- загрузка собственных символов в индикатор

#### TDevLCD\_HD44780x1

Данный класс наследуется от класса TDevLCD\_HD44780\_Base и реализует следующие методы:

Tick	- запись данных в индикатор (код выполняется в прерывании 1 ms)
------	---

#### TDevLCD\_HD44780x2

Данный класс наследуется от класса TDevLCD\_HD44780\_Base и реализует следующие методы:

Tick	- запись данных в индикатор (код выполняется в прерывании 1 ms)
------	---

#### TDevLCD\_HD44780\_TypeXXXX

Данная серия классов определяет параметры индикатора, а также производит выборку команд управления. Класс реализует следующие методы:

GetData	- выборка команд управления индикатором (код команды установки адреса начала строки)
---------	--

#### TDevLCD\_HD44780\_XXXX

Данная серия классов упрощает определение класса работы с индикатором за счёт уменьшения количества шаблонных параметров передаваемых в класс TDevLCD\_HD44780x1. Класс не имеет методов.

Рассмотрим порядок записи данных в индикатор.

Параметры индикатора на 16 символов и 2 строки описываются следующим образом:

```
lib\device\TDevLCD_HD44780_Type.h
```

```
31 | static const uint8_t __flash TableDevLCD_HD44780_Type1602[] =
32 | {
33 |     0x80, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
34 |     0xC0, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F
35 | };
...
143 | struct TDevLCD_HD44780_Type1602
144 | {
145 |     enum { LCD_COL = 16, LCD_ROW = 2, LCD_SIZE = (LCD_COL) * (LCD_ROW) };
146 |     //-----
147 |     #pragma inline = forced
148 |     static inline uint8_t GetData(const uint8_t value)
149 |     {
150 |         return (TableDevLCD_HD44780_Type1602[value]);
151 |     }
152 | };
```

В строке 31 определяем массив байт с командами управления индикатором. Массив размещён во Flash, о чём говорит ключевое слово \_\_flash.

В строках 33 и 34 первые байты (0x80 и 0xC0) представляют собой команды установки адреса начала строк. Далее следует порядковый номер символа в буфере индикатора для отображения. В зависимости от типа индикатора данные команды установки адреса могут меняться, перед использованием необходимо уточнить в datasheet параметры индикатора.

В строке 143 определяем метод для выборки команды управления индикатором из массива байт.

В строке 145 фиксируем параметры индикатора.

В строке 150 производим выборку команды из массива.

Запись данных в индикатор производится в прерывании 1 ms. Код, реализующий запись одного байта в индикатор, приведён ниже:

lib\device\TDevLCD\_HD44780.h

```

96 template < class Driver , class Type >
97 class TDevLCD_HD44780x1 : public TDevLCD_HD44780_Base < Driver, Type >
98 {
99     STATIC_ASSERT((Driver::HD44780 == 1), "TDevLCD_HD44780x1: Type of the driver not correct");
100
101     public:
102     //-----
103     // Код, выполняемый в прерывании 1 мксек (отправка одного байта)
104     //-----
105     #pragma inline = forced
106     static inline void Tick(void)
107     {
108         typedef TDevLCD_HD44780_Base < Driver, Type > base;
109
110         if(base::BufferCount > (Type::LCD_SIZE + Type::LCD_ROW))
111             base::BufferCount = 0;
112         uint8_t cmd = Type::GetData(base::BufferCount);
113         if(BIT_TST(cmd, 7))
114         {
115             Driver::SetModeCmd();
116             Driver::Write(cmd);
117             Driver::SetModeData();
118         }
119         else
120         {
121             uint8_t data = base::Buffer[cmd];
122             // data = TableDevLCD_HD44780_CharDecode[data];
123             Driver::Write(data);
124         }
125         base::BufferCount++;
126     }
127 };

```

В строке 96 указаны параметры шаблонного класса TDevLCD\_HD44780x1.

В строке 97 класс TDevLCD\_HD44780x1 наследуется от класса TDevLCD\_HD44780\_Base.

В строке 99 производится проверка на количество контроллеров HD44780, поддерживаемых драйвером.

В строке 108 для сокращения длины имени класса определяем его короткий псевдоним.

В строках 110, 111 производится проверка счётчика переданных байт на достижение конца буфера индикатора и установка счётчика на начало при достижении конца буфера.

В строке 112 считываем из массива код команды.

В строке 113 проверяем старший бит. Если бит установлен, то переменная data содержит команду, которую необходимо отправить в индикатор для перехода на следующую строку. Если бит сброшен, то переменная data содержит порядковый номер символа, выводимого на индикатор.

В строке 115 устанавливаем для индикатора режим записи команд.

В строке 116 записываем байт в индикатор.

В строке 117 устанавливаем для индикатора режим записи данных.

В строке 121 переменная data перезаписывается кодом символа для вывода на индикатор.

В строке 122 (закомментирована) производится конвертация символа из кодировки CP1251 в кодировку индикатора.

В строке 123 записываем байт в индикатор.

В строке 125 увеличиваем счётчик байт переданных в индикатор.

#### 4.4 Работа с буфером индикатора

Для работы с буфером индикатора создан шаблонный класс TDisplayChar. В качестве шаблонного параметра в класс TDisplayChar передаётся класс устройства «LCD-индикатор» (DevLCD).

Класс TDisplayChar является шаблонным классом и определяется следующим образом:

mcu\TFlagList.h

```
3 template < class Device >
4 class TDisplayChar
```

где,

class Device – класс устройства вывода (индикатор)

Для работы с классом TDisplayChar необходимо создать устройство, реализующее определённый интерфейс. Устройство должно поддерживать следующие публичные переменные и методы:

Buffer	- переменная (массив байт), буфер индикатора
SetPos	- установка позиции курсора

Класс TDisplayChar реализует следующие методы

Clear	- очистка дисплея
SetPos	- установка позиции курсора в заданную позицию
ShowChar	- вывод одного символа с заданной позиции
ShowText	- вывод строки на индикатор с заданной позиции
ShowValue	- вывод целого беззнакового числа с заданной позиции
ShowTextValue	- вывод строки и беззнакового значения с заданной позиции

Методы всегда могут быть добавлены для расширения возможностей работы с индикатором.

## 4.5 Настройка и использование

Пример настройки и использования приведён ниже:

main.h

```
40 typedef TDevLCD_HD44780_1602 < TDrvLCD_HD44780_8bit > DevLCD;
...
43 typedef TDisplayChar < DevLCD > Display;
```

main.cpp

```
7 const uint8_t __flash MsgAttention[] = LCD_TEXT_08("Внимание");
...
12 void main(void)
13 {
...
19     DevLCD::Init();
...
27     Display::Clear();
28     Display::ShowText(Display::SetPos(0,0),MsgAttention);
...
50     if(Counter_20Hz::Check())
51     {
52         Display::ShowValue(Display::SetPos(0,12),value16,4,' ');
53         Display::ShowValue(Display::SetPos(1,0),value8,3,' ');
54         Counter_20Hz::Reset();
55     }
56 }
57 }
```

IRQ\_1ms.h

```
12 struct IRQ_1ms
13 {
14     #pragma inline = forced
15     static inline void execute(void)
16     {
...
20         DevLCD::Tick();
...
22     }
23 };
```

### Файл main.cpp

В строке 19 выполняем инициализацию индикатора.

В строке 27 очищаем дисплей. Буфер дисплея заполняется пробелами.

В строке 28 выводим из Flash на дисплей сообщение «Внимание».



В строке 50 проверяем событие «Counter\_20Hz».

В строке 52 выводим значение переменной value16 (строка 0, позиция 12).

В строке 53 выводим значение переменной value8 (строка 1, позиция 0).

В строке 54 сбрасываем признак «Counter\_20Hz». Событие обработано.

#### Файл IRO\_1ms.h

В строке 20 выполняем запись в индикатор одного байта.

При переходе от 8-проводной схемы подключения индикатора к 4-проводной достаточно заменить драйвер устройства:

1	<b>typedef</b> TDevLCD_HD44780_1602 < TDrvLCD_HD44780_8bit > DevLCD;
1	<b>typedef</b> TDevLCD_HD44780_1602 < TDrvLCD_HD44780_4bit > DevLCD;

## 4.6 Конвертирование текста

Данный раздел не имеет отношения к языку C++. Но так как речь идёт о символьном LCD-индикаторе, то представленная ниже информация может быть полезной.

Кодировка LCD-индикатора на контроллере HD44780 не соответствует кодировке исходных текстов (как правило, это CP1251). Поэтому, для корректного отображения текста на индикаторе его необходимо конвертировать. Существует два основных способа конвертирования текста:

- конвертировать силами микроконтроллера через таблицу;
- предварительно преобразовать и присвоить значение глобальной переменной (константе).

Первый способ довольно простой, но требует наличия во Flash таблицы перекодировки и занимает ресурсы микроконтроллера. Для современных микроконтроллеров это уже не является проблемой, но всё же это бесполезная работа для микроконтроллера. Использование таблицы упрощает жизнь разработчику, но усложняет микроконтроллеру.

Реализовать конвертирование можно следующим образом:

<b>lib\device\TDevLCD_HD44780.h</b>	
7	<b>static const uint8_t</b> __flash TableDevLCD_HD44780_CharDecode[] =
8	{
9	0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
10	0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
11	0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
12	0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
13	0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x49, 0x4A, 0x4B, 0x4C, 0x4D, 0x4E, 0x4F,
14	0x50, 0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58, 0x59, 0x5A, 0x5B, 0x20, 0x5D, 0x5E, 0x5F,
15	0x60, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67, 0x68, 0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F,
16	0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78, 0x79, 0x7A, 0x20, 0x20, 0x20, 0x20, 0x20,
17	
18	0xD9, 0xDA, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
19	0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
20	0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0xA2, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
21	0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0xB5, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
22	0x41, 0xA0, 0x42, 0xA1, 0xE0, 0x45, 0xA3, 0xA4, 0xA5, 0xA6, 0x4B, 0xA7, 0x4D, 0x48, 0x4F, 0xA8,
23	0x50, 0x43, 0x54, 0xA9, 0xAA, 0x58, 0xE1, 0xAB, 0xAC, 0xE2, 0xAD, 0xAE, 0x62, 0xAF, 0xB0, 0xB1,
24	0x61, 0xB2, 0xB3, 0xB4, 0xE3, 0x65, 0xB6, 0xB7, 0xB8, 0xB9, 0xBA, 0xBB, 0xBC, 0xBD, 0x6F, 0xBE,
25	0x70, 0x63, 0xBF, 0x79, 0xE4, 0x78, 0xE5, 0xC0, 0xC1, 0xE6, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7
26	};
	...
106	<b>static inline void</b> Tick( <b>void</b> )
107	{
	...
121	uint8_t data = base::Buffer[cmd];
122	data = TableDevLCD_HD44780_CharDecode[data];
123	Driver::Write(data);
	...
126	}

В строке 121 производится выборка очередного символа из буфера индикатора в кодировке CP1251.

В строке 122 производится конвертирование символа в кодировку индикатора согласно таблице TableDevLCD\_HD44780\_CharDecode[].

В строке 123 производится запись символа в индикатор.

Код метода Tick выполняется в прерывании 1 ms.

Второй способ сложнее. Проблема в том, что нельзя для глобальных переменных использовать функции (методы). Поэтому единственный способ конвертировать текст без накладных расходов это либо преобразовать заранее (например, сторонними программами), либо попытаться доверить эту работу препроцессору языка Си (C++).

Преобразование текста вручную не совсем наглядно, нужно было придумать другой способ преобразования. В итоге получился довольно большой макрос (файл TDevLCD\_HD44780\_Conv.h). Макрос конвертирует текст только из кодировки CP1251 в кодировку LCD-индикатора на контроллере HD44780.

Макрос начинается с префикса LCD\_TEXT\_XX, где XX это количество символов в строке (отсчёт начинается с единицы).

Например, если нам необходимо вывести на индикатор слово «Внимание», то макрос будет выглядеть так:

```
LCD_TEXT_08 ("Внимание")
```

где «08» это количество символов в слове «Внимание».

Пример использования:

main.cpp

```
7 | const uint8_t __flash MsgAttention[] = LCD_TEXT_08("Внимание");
...
28 | Display::ShowText(Display::SetPos(0,0),MsgAttention);
```

#### 4.7 Результаты компиляции проекта

Для эмуляции в Proteus для данного проекта создано три схемы:

- ATtiny2313 + LCD1602;
- ATmega32 + LCD1602;
- ATmega32 + LCD4004.

Т.к. для подключения периферии к микроконтроллерам используются одинаковые имена портов ввода/вывода, то корректировка исходных текстов не требуется (за исключением имени вектора прерывания таймера 0). Выбор микроконтроллера производится в настройках проекта согласно п.1.3.

Результат компиляции проекта на ATtiny2313 + LCD1602 (модель памяти Tiny):

Memory	4 битная шина LCD-индикатора		8 битная шина LCD-индикатора	
	Оптимизация High + Speed	Оптимизация High + Size	Оптимизация High + Speed	Оптимизация High + Size
Flash, байт	1 332	1 198	1 112	1 102
RAM, байт	107			

Результат компиляции проекта на ATmega32 + LCD1602 (модель памяти Small):

Memory	4 битная шина LCD-индикатора		8 битная шина LCD-индикатора	
	Оптимизация High + Speed	Оптимизация High + Size	Оптимизация High + Speed	Оптимизация High + Size
Flash, байт	1 400	1 348	1 180	1 226
RAM, байт	107			

Результат компиляции проекта на ATmega32 + LCD4004 (модель памяти Small):

Memory	4 битная шина LCD-индикатора		8 битная шина LCD-индикатора	
	Оптимизация High + Speed	Оптимизация High + Size	Оптимизация High + Speed	Оптимизация High + Size
Flash, байт	1 568	1 454	1 284	1 304
RAM, байт	235			