Библиотека AVR++ v0.5beta

Назначение, идеи, руководство пользователя

ELLION ПОЛУКОНДУКТОРЗ ОЙЪ

25 марта 2011 г.

Илья Меркин

Библиотека AVR++ v0.5beta

Назначение, идеи, руководство пользователя

ВНИМАНИЕ! В ДАННОЙ ВЕРСИИ БИБЛИОТЕКИ РУКОПИСНАЯ ДОКУМЕНТАЦИЯ НЕПОЛНА. ОНА ОБЪЯСНЯЕТ ИДЕЮ БИБЛИОТЕКИ И ВЫБРАННЫЙ ПОДХОД, НО В ДАЛЬНЕЙШЕМ НЕОБХОДИМО ЗНАКОМИТЬСЯ С ДЕМОНСТРАЦИОННЫМИ ПРОГРАММАМИ, КОТОРЫЕ РАСПОЛОЖЕНЫ В КАТАЛОГЕ samples АРХИВА С БИБЛИОТЕКОЙ.

От автора

Эта библиотека возникла достаточно неожиданным для меня образом. После многих лет программирования, в основном на C++, во мне неожиданно проснулся электронщик, которым я являюсь по образованию (давным-давно заканчивал Военмех в Санкт-Петербурге). Вполне естественно, что программисту интересны микроконтроллеры, так что я увлёкся AVR'ами и начал проектировать устройства на них, просто в качестве хобби. Писал на ассемблере и C, но в какой-то момент обнаружил проект scmRTOS и с удивлением понял, что C++ для микроконтроллеров тоже не лишён смысла. Более того, именно здесь, в условиях жёстких ограничений, можно обоснованно применить массу интересной экзотики, которую в прикладном программировании больших проектов (то, чем я занимаюсь) задействовать очень и очень проблематично. Например, я без восторга отношусь к широкому применению шаблонов имени Александреску, но именно тут они показались мне очень полезными.

Для чего предназначена библиотека

В первую очередь – для работы с GPIO. Казалось бы, для этого вполне достаточно существующих в avr-gcc макросов **PORTA**, **PORTB**, **PORTC**... Где тут простор для создания библиотеки, спрашивается? Вот об этом и поговорим для начала.

Пусть Вы макетируете устройство и достаточно часто меняете схему подключения периферии. Скажем, есть у Вас LCD на контроллере HD44780, и Вы подключили его восьмибитную шину данных к **PORTC** ATmega32, а через неделю вдруг решили, что Вам нужен JTAG, отнимающий несколько пинов порта **C**. Если Ваш прикладной код работает с портами ввода и вывода напрямую, он потребует модификации, причём, возможно, достаточно содержательной: когда нет свободного восьмибитного порта, подключать LCD придётся «вразнотык» к тем выходам, которые свободны. Выставлять значения сигнала на выходе тоже придётся не одним присваиванием, а множеством установок отдельных битов. Как следствие, код придётся сильно править.

Самое обидное, что такая же ситуация возникает не только при модификации устройства, но и при попытке повторного использования кода. В некоторых библиотеках на C (скажем, я видел такую библиотеку для работы с HD44780) проблему решают с помощью многочисленных макроопределений. Но это – не самый удобный метод для кода на C++.

Во-первых, если уж код пишется с обширным использованием макроопределений, то это не совсем C++, и почему бы просто не писать на старом добром C. Во-вторых, сделать на макроопределениях можно далеко не всё, и именно это я хотел показать в библиотеке. В-третьих, современные компиляторы C++ (и avr-gcc не является исключением) давно научились действительно качественно работать с шаблонами: если ещё во второй половине 90-х годов даже использование STL требовало определённой осторожности, то сейчас шаблонные списки типов из нескольких сот элементов успешно обрабатываются и могут быть использованы для порождения работоспособного и эффективного кода.

Основные идеи библиотеки и выбранный подход

Осторожно: приведённые в этой маленькой главке примеры кода зачастую сильно упрощены, из-за чего могут не быть вполне корректными на реальной библиотеке. Это именно описание того, что хотелось сделать, бывшее неформальным изначальным ТЗ в процессе разработки. В реальности некоторые вещи оказалось разумным сделать немного по-другому, и если Вы хотите образцов реального кода, загляните в каталог samples, там есть кое-что. Позже, в главе «Руководство программиста» мы разберём некоторые примеры. ☺

Традиционно язык C++ ассоциируется у большинства с объектно-ориентированным программированием. Во многом это правильно, но в библиотеке AVR++ ООП нам практически не пригодится, поскольку мы будем избегать создания объектов и накладных расходов на их создание. Вместо этого мы будем пытаться использовать техники порождающего программирования на шаблонах C++ для того, чтобы на этапе компиляции генерировать эффективные алгоритмы работы с наборами битов, причём выполняться эти алгоритмы будут статическими методами классов, т.е. без привязки к объекту. С точки зрения пользователя библиотеки AVR++ её код является процедурным. Настройки будут вынесены в параметры шаблонов классов, и цель этих шаблонов – предоставить эффективные реализации статических методов для конкретных наборов параметров.

Работа с портами МК

Работа с портами ввода и вывода, сводящаяся к выставлению битов, может быть реализована средствами шаблонов C++, параметризуемых портом и номером бита внутри порта. При этом работа с битом порта осуществляется не объектом класса, а самим классом (используются статические методы). Как следствие, объекты не создаются, и при включённой оптимизации какие-либо накладные расходы по сравнению с кодом на C отсутствуют. Операции выглядят примерно так:

```
typedef output<port_a,2> my_pin; //PORTA, bit 2.
my_pin::config(); // Set data direction (I/O to O)
my_pin::set(true); // Set bit to 1.
```

В дальнейшим классами пинов можно будет параметризовать код, работающий с периферией. В качестве примера можно привести данное несколько ниже описание порта расширения.

Работа с «расширителями портов» («виртуальными портами»)

Работа с расширителями портов (например, регистрами с последовательной загрузкой данных) может быть реализована так, что пользователь при написании кода не почувствует никакой разницы между обычным выходным пином порта МК и «виртуальным пином», подключённым более сложным образом.

Так, я регулярно использую сдвиговые регистры с защёлкой 74HC595. Этот регистр представляет собой 16-ногую микросхему и позволяет последовательно загружать данные бит за битом, а потом выдавать эти данные на параллельные выходы по команде от МК. При этом поддерживается объединение регистров в цепочки, за счёт чего можно получать многобайтовые порты любой разрядности, кратной восьми. Для управления такой цепочкой регистров хватает всего лишь трёх ножек МК. Вот как-то так хотелось бы определять порт и пины внутри него:

```
typedef output<port_a,2> clk_pin; //74HC595 clock pin
typedef output<port_a,3> data_pin; //74HC595 data pin
typedef output<port_b,7> latch_pin; //74HC595 clock pin
// Now let's define 4-byte extension port;
typedef extension_out_port<clk_pin,data_pin,latch_pin,4> xport;
xport::pin<xport,2,7> my_pin; //byte 2, bit 7
DECLARE_XPORT_DATA(xport); //Declare port data buffer


xport::init ();
my_pin::set(true); //Set bit to 1
my_pin::port::update_outputs(); // To be discussed further
```

Впоследствии **my_pin** можно использовать ВЕЗДЕ, где можно использовать обычные пины. Если Ваша библиотека для работы, допустим, с LCD, определяет шаблонные классы, параметризуемые классами пинов библиотеки AVR++, переключение устройства с реальных портов вывода на расширения должно быть тривиальным.

Оптимизация работы с «логическими шинами»

В случае использования расширителей особенно остро встаёт вопрос об эффективности кода. Так, если Вы управляете 16-битным портом расширения, и Вам необходимо установить 8 из 16 битов порта, это нельзя делать «в лоб», просто последовательно выполняя обновление каждого из 8 битов. Дело в том, что после обновления необходимо повторно загружать все 16 битов регистра, и важно понять, что повторную загрузку можно выполнить один раз, а не после выставления каждого бита. Особенно интересной задача становится в том случае, если биты, которыми Вы хотите управлять совместно, не являются битами одного порта. После установки битов надо обновить все затронутые порты, причём желательно по одному разу.

Для решения этой проблемы в AVR++ операция обновления данных разделена на два этапа: собственно установку бита и обновление порта. Для обычных портов ввода-вывода операция обновления пуста, однако для виртуальных портов именно в этой операции производятся те действия, которые необходимо отложить до момента, когда будет установлен последний бит. В приведённом ранее примере это загрузка данных в цепочку сдвиговых регистров и подача сигнала на защёлку. Поэтому в предыдущем примере и есть загадочная строка:

```
my_pin::port::update_outputs(); // To be discussed further
```

Последней деталью в оптимизации ввода-вывода являются логические шины. Биты, объединённые логически и часто обновляемые совместно (например, биты шины данных LCD), но при этом физически, возможно, не располагающиеся рядом, целесообразно объединять в логические шины:

```
typedef extension_out_port<clk_pin,data_pin,latch_pin,4> xport;
DECLARE_XPORT_DATA(xport); //Declare port data buffer

typedef output<port_c,2> p0;
typedef output<port_d,3> p1;
typedef xport::pin<xport,2,6> p2;
typedef xport::pin<xport,2,7> p3;

typedef TLIST4(p3,p2,p1,p0) pin_list; // A type list

typedef output_bus<pin_list> my_bus;

my_bus::config ();
my_bus::out (0x0f); // Set all the bits to 1.
my_bus::update_outputs ();
```

В этом примере логическая шина разбросана по выводам: задействованы PC2, PD3 и два бита из второго байта расширителя порта `xport`. Что важно, при обновлении данных расширитель будет обновлён всего один раз. Оптимальный алгоритм обновления данных формируется на этапе компиляции, и, помимо логики повторной загрузки данных в расширители, может оптимизировать также параллельную установку битов, взаимное расположение которых в логической шине и в выходном регистре одинаково.

Политики задержек и политики времени

Реализация задержек в коде часто становится проблемой, и проблем на самом деле две, а не одна: выбор величины задержек и выбор механизма реализации задержек. Очевидно, что проблемы эти ортогональны и должны решаться раздельно.

Политики времени (timing policies)

В некоторых случаях Вам может требоваться изменение величин задержек, необходимых для работы с каким-либо внешним устройством. Допустим, есть у Вас библиотека для работы с 74HC595, а надо подключить 74xxx595, где xxx – всё, что угодно, только не HC. В этом случае логика работы микросхемы остаётся прежней, а вот величины требуемых задержек (например, время, в течение которого следует удерживать бит синхронизации) могут отличаться. Хочется иметь возможность написать код работы с микросхемой так, чтобы его легко можно было перенастроить на новый набор задержек. Для этого в AVR++ предусмотрен механизм политик времени. Каждая такая политика – это класс, содержащий информацию о величинах задержек для решения той или иной задачи, и такой класс можно использовать для настройки библиотеки.

Политики задержек (delay policies)

Во-вторых, сам способ обеспечения задержки может быть различен. В некоторых случаях задержка обеспечивается простым циклом, иногда вызывается функция стандартной библиотеки (в avrlibc это `_delay_ms` или `_delay_us`), а иногда, например, в многозадачной операционной системе реального времени, Вам придётся делать сложные трюки. Допустим, для малых задержек можно использовать простой цикл (с запретом или без запрета прерываний), а вот большие придётся обеспечивать путём вызовов API операционной системы. Для того, чтобы код, реализующий работу с оборудованием, не был завязан на конкретный способ реализации задержек, в AVR++ предусмотрены политики задержек – классы, заключающие в себе способы отработки задержек разной величины.

Когда всё это действительно нужно?

Изначально я позиционирую AVR++ как библиотеку для написания библиотек. Это не значит, что более нигде AVR++ пригодиться не может, просто именно в повторно используемом коде она будет максимально полезна, поскольку именно такой код чаще всего не знает заранее, к каким битам каких портов будут подключены управляемые им периферийные устройства. Согласитесь, заманчиво написать библиотеку для работы с LCD, которая впоследствии может допускать подключение LCD не только напрямую к МК, но и к расширителям портов, причём абсолютно прозрачным для себя способом, никак не утруждая себя знаниями о внутреннем устройстве подобных расширителей?

На первый взгляд кажется, что такая библиотека сама будет шаблонной и полностью заголовочной, но в действительности это не совсем так. Можно применить стандартный для микроконтроллерных библиотек трюк: библиотека компилируется как отдельный модуль, но включает в себя файл конфигурации, определяющий подключение нужного устройства. В этом случае можно уйти от использования шаблонов в коде библиотеки работы с периферией и вынести её в отдельную единицу трансляции, что иногда является большим плюсом.

Недостатки подхода

Вообще говоря, мне известен только один недостаток, при определённых условиях могущий стать важным. При применении описанного подхода необходимо включать оптимизацию компилятора, иначе Вы рискуете утонуть в диких «шаблонизмах», вызовах пустых функций и прочей дребедени, которую при включённой оптимизации компилятор уничтожает «на ура», зачастую сводя выставление бита порта к одной ассемблерной команде **`SBI`** или **`CBI`**. Но известно, что включённая оптимизация часто мешает отладке. И хотя при достаточно малых ресурсах серии AVR оптимизацию всё равно на определённом этапе роста проекта придётся включать, чтобы тривиально влезть в память, есть шансы, что в коде, использующем AVR++, оптимизацию придётся включить несколько раньше.

Для примера: когда набор тестов библиотеки при выключенной оптимизации уже вышел за размер памяти ATmega16 (то есть за 16K), он же, но с включённой оптимизацией по размеру кода, занимал всего 3K, то есть в 5 раз меньше.