Роман Абраш
г. Новочеркасск
E-mail: **arv@radioliga.com**

Книга по работе с WinAVR и AVR Studio

Продолжение. Начало в №1-6/2010

Наблюдение за ресурсами проекта

Для наблюдения за содержимым всех ресурсов микроконтроллера и переменных в программе пользователя во время ее отладки AVR Studio предоставляет богатый набор средств. Прежде всего, это «Окно периферии», предоставляющее удобный интерфейс наблюдения и изменения состояния всех регистров управления встроенными периферийными устройствами микроконтроллера. Во время отладки состояние отдельных битов изображается закрашенными в черный цвет квадратиками; щелкнув на любом из них, программист может изменить состояние бита на противоположное. Это бывает необходимо в следующих случаях:

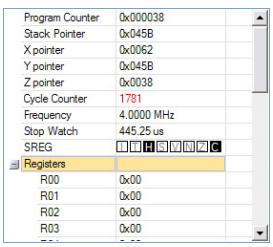
- обнаружена ошибка в программе, заключающаяся в «инверсном» анализе какого-либо бита (т.е. надо проверять на равенство 1, а в программе ошибочно проверяется на 0 и т.п.). Конечно, можно остановить процесс отладки и, исправив ошибку, перекомпилировать программу, однако часто удобнее принудительно изменить бит в регистре «наоборот», чтобы «обмануть» неверную программу, заставив ее правильно отреагировать на ситуацию, чтобы продолжить отладку остальных участков кода;

- AVR Studio не поддерживает корректную эмуляцию периферийного устройства контроллера. Например, невозможна эмуляция АЦП – значение в регистрах результата AVR Studio никогда самостоятельно не изменяет, и для того, чтобы проимитировать факт реального измерения, программист должен самостоятельно ввести значения в соответствующие регистры;

- необходимо проимитировать поступление на порт микроконтроллера сигнала извне (от прочих элементов схемы). В этом случае нужно изменить значение соответствующего регистра PINx вручную. Кроме ручного способа имеется и «полуавтоматический», так называемое «стимулирование порта», которое рассматривается в следующей главе более подробно.

Кроме окна периферии имеется и еще ряд окон и панелей, управляемых при помощи меню «**View**». Рассмотрим их подробно.

Панель состояния процессора – **Processor**. По умолчанию (если расположение панелей не было изменено) в режиме отладки автоматически активируется в области, где находится и окно проекта:



В этой панели отображаются как некоторые недоступные для принудительного изменения значения, так и доступные.

Program Counter – программный счетчик, показывает значение PC, т.е. в сущности адрес очередной исполняемой команды.

Stack Pointer – указатель стека, показывает адрес ОЗУ, хранящийся в регистре SP.

X pointer, **Y pointer** и **Z pointer** – показывает значения указателей X, Y и Z.

Рассмотренные три параметра недоступны для принудительного изменения в ходе отладки.

Cycle Counter – счетчик машинных циклов, показывает число тактов, потраченных на исполнение всех команд с момента старта программы. Этот счетчик может быть сброшен в любое время [28].

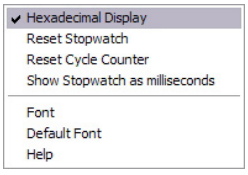
Frequency – тактовая частота, соответствует значению, заданному в настройках эмулятора (см. предыдущую главу).

Stop Watch – время остановки. Это значение показывает время, прошедшее с момента начала исполнения программы (т.е. с момента старта отладки) до момента ее приостановки. Этот «секундомер» может быть сброшен пользователем в любой момент, что позволяет засекать время исполнения отдельных участков программы.

SREG – содержимое регистра статуса ядра микроконтроллера, показывает состояние всех битов этого регистра, которые доступны для изменения пользователем.

Далее следует группа регистрового файла микроконтроллера – **Registers**, в которой показано содержимое всех 32 регистров. Эти значения так же доступны для модификации в любое время.

Для панели **Processor** доступно всплывающее меню:



Hexadecimal Display – если отмечено, то все или только выбранное мышкой значение будет отображаться в шестнадцатеричном формате, в противном случае используется десятичный формат.

Reset Stopwatch – сброс времени остановки (сброс «секундомера»).

Reset Cycle Counter – сброс счетчика машинных тактов.

Show Stopwatch in milliseconds – время остановки отображать в миллисекундах (по умолчанию счет в микросекундах).

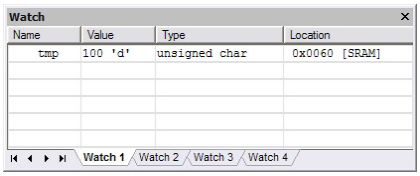
Font – изменить шрифт, используемый для вывода содержимого панели.

Default Font – установить для панели шрифт по умолчанию.

Help – вызов справки (на английском) о панели.

[28] Здесь и далее под «любым временем» подразумевается любой момент остановленного исполнения программы. Во время автоматического или автоматического пошагового исполнения все средства изменения состояния регистров и портов недоступны.

Окно-панель наблюдения за переменными **Watch:**



Это окно содержит 4 закладки, в каждой из которых можно наблюдать и при необходимости модифицировать содержимое любых переменных (в том числе регистров микроконтроллера) во время отладки. Информация представлена в виде таблицы из 4-х столбцов:

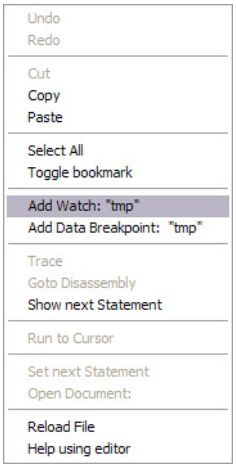
Name – имя переменной. Можно ввести имя переменной вручную, выполнив двойной щелчок в первой свободной строке. Двойной щелчок на имени уже имеющемся в окне позволяет изменить его, т.е. выбрать другую переменную для наблюдения.

Value – значение переменной. Показывается числовое и, если возможно, символьное представление. Выполнив двойной щелчок в этом столбце, можно принудительно изменить значение переменной, введя любую допустимую в Си константу.

Type – тип переменной.

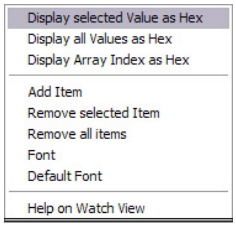
Location – адрес начала области памяти, выделенной для хранения переменной. В квадратных скобках указывается тип памяти (встроенное ОЗУ или внешнее).

Кроме ручного ввода имени переменной, в окно Watch можно перетащить и бросить идентификатор переменной прямо из текста программы, т.е. выделить переменную, «схватить» ее и перенести в это окно. Есть и третий способ – установив курсор на интересующую переменную в тексте программы, щелкнуть правой кнопкой мыши и в появившемся меню (см. рисунок) выбрать команду **Add Watch**.



Наконец, аналогичный результат достигается и при нажатии на кнопку  на панели кнопок.

Для окна **Watch** имеется всплывающее меню:



Display selected Value as Hex – отображать выбранное

значение (необходимо предварительно выделить строку в таблице) в виде шестнадцатеричного числа.

Display all Values as Hex – все значения показывать в шестнадцатеричном формате.

Display Array Index as Hex – индексы массивов показывать в шестнадцатеричном формате.

Add Item – добавить переменную

Remove selected Item – удалить из окна выделенную строку (переменную)

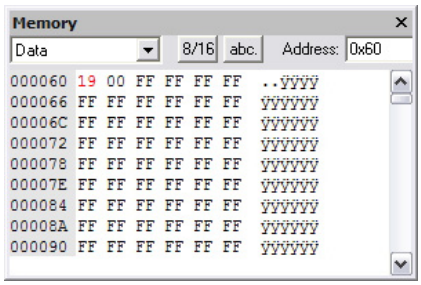
Remove all items – удалить из закладки все переменные

Font и Default font – изменение шрифта для окна, как уже было сказано ранее.

Help on Watch View – вызов справки об окне.

К сожалению, просмотр массивов в окне Watch на протяжении многих версий AVR Studio сопровождается одним неудобством: содержимое массива не обновляется в момент остановки программы (на точке останова или принудительно), поэтому приходится «свернуть» и затем «развернуть « массив, чтобы увидеть его актуальное содержимое. Для сворачивания и разворачивания массивов в соответствующей строке окна будет находиться кнопочка с «плюсиком» или «минусом» соответственно.

Окно просмотра содержимого памяти **Memory** [29]:



Если окно **Watch** позволяет наблюдать и модифицировать значения переменных, то данное окно позволяет аналогично оперировать содержимым любых ячеек памяти безотносительно к их распределению по переменным.

В верхней части окна имеется ряд органов управления:

· Список типов наблюдаемой памяти: **Data** (ОЗУ данных), **EEPROM**, **I/O** (область портов), **Program** (Flash память программ) и **Register** (область адресов регистрового файла).

· Кнопка **8/16**, позволяющая изменить разрядность отображаемых данных – 8 или 16 бит.

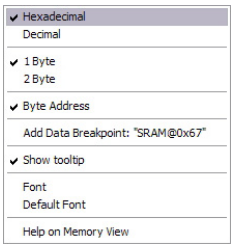
· Кнопка **abc**, включающая или отключающая показ символьного представления содержимого.

· Поле **Address**, задающее адрес первой отображаемой ячейки в окне.

Если на очередном шаге отладки содержимое ячейки памяти изменилось – это выделяется красным цветом.

[29] Как было упомянуто ранее, таких окон может быть до трех.

При помощи команд всплывающего меню можно гибко управлять как отображением, так и содержимым наблюдаемой области памяти:



Hexadecimal и **Decimal** позволяют переключить формат вывода содержимого памяти.

1 Byte или **2 Byte** переключают разрядность данных (как и кнопка **8/16**).

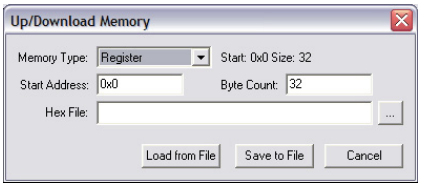
Byte address переключает режим вычисления адреса – побайтно (если отмечено) или по 16-битным словам.

Add Data Breakpoint – команда установки точки остановки по изменению содержимого указанной ячейки памяти.

Show tooltip – включает или отключает всплывающие подсказки.

Вы можете изменить произвольно выбранную ячейку и посмотреть, как это скажется на работе вашей программы. Очень удобно при помощи этого окна определять глубину стека, необходимую для работы программы: запускаем программу на исполнение, ждем достаточное время для того, чтобы все ветви алгоритма отработали, а затем останавливаем программу и открываем окно просмотра ОЗУ. Будет хорошо видно, что в начале области памяти и в конце ячейки содержат какие-то значения – в начале область переменных, а в конце область, использованная стеком. Если между этими областями имеется достаточное количество пустых ячеек (содержат значение 0xFF) – все нормально, стек не затирает область переменных. Если между этими областями нет пустоты или всего две-три ячейки не заняты – это очень тревожный признак –скорее всего такая программа в реальности работать не будет из-за переполнения стека.

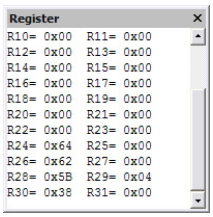
В комплексе с окном **Memory** удобно использовать другую возможность – загрузку или сохранение содержимого области памяти из/в файла, реализуемую командой **Up/Download Memory** из меню «**Debug**». В этом случае появляется окно следующего вида:



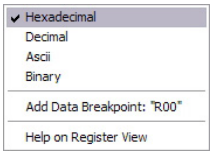
Точно так же вы можете указать тип памяти – список **Memory Type**, задать адрес первой обрабатываемой ячейки **Start Address**, количество обрабатываемых ячеек **Byte Count** (при этом ориентируйтесь на подсказку выше – значения **Start** и **Size**, которые определяют границы выбранной области). В поле **Hex File** нужно указать имя файла, с которым будет осуществлена работа. Кнопка **Load from File** позволяет загрузить из указанного файла содержимое в выбранную область памяти, а кнопка **Save to File** выполняет обратную операцию – сохраняет указанную область в заданный файл. Формат файла – *Intel HEX*. Таким образом, реализуется достаточно удобный механизм работы с «загружаемыми» данными.

Вернемся к окнам наблюдения: очередное из них – это окно просмотра регистрового файла **Register**.

Оно дублирует содержимое регистров в панели **Processor**, а по функциональности соответствует только что рассмотренным окнам с той лишь разницей, что



позволяет отображать информацию с большим разнообразием форматов, указываемых через всплывающее меню:



В дополнению к шестнадцатеричному и десятичному форматам, здесь имеется возможность указать символьный (**Ascii**) или двоичный (**Binary**).

Последнее окно, нередко необходимое для отладки, это окно дизассемблера (**Disassembler**). Это окно располагается обычно в основной области, т.е. там же, где и исходный текст. В нем выводится дизассемблированный код программы, т.е. восстановленный до команд ассемблера. При этом операторы Си так же показаны (что позволяет увидеть, какими ассемблерными командами реализован тот или иной оператор Си):

```
+0000002B:   F7E1        BRNE    PC-0x03          Branch if not equal
+0000002C:   D002        RCALL   PC+0x0003        Relative call subroutine
+0000002D:   C018        RJMP    PC+0x0019        Relative jump
+0000002E:   CFD1        RJMP    PC-0x002E        Relative jump
@0000002F: main
---- demo1.c ----------------------------------------------------------------
5:          int main(void){
+0000002F:   93CF        PUSH    R28              Push register on stack
+00000030:   93DF        PUSH    R29              Push register on stack
+00000031:   B7CD        IN      R28,0x3D         In from I/O location
+00000032:   B7DE        IN      R29,0x3E         In from I/O location
6:          DDRB = 255;
+00000033:   E3E7        LDI     R30,0x37         Load immediate
+00000034:   E0F0        LDI     R31,0x00         Load immediate
+00000035:   EF8F        SER     R24              Set Register
+00000036:   8380        STD     Z+0,R24          Store indirect with displacement
7:          while (tmp){
+00000037:   C005        RJMP    PC+0x0006        Relative jump
```

Содержимое окна дизассемблера напоминает содержимое файла-листинга, за исключением того, что формируется не компилятором, а AVR Studio.

Имитация входных сигналов и наблюдение выходных

Микроконтроллер, не смотря на всю его многофункциональность, так или иначе взаимодействует с остальными элементами схемы конкретного устройства, т.е. должен реагировать на входные сигналы, формируя выходные. Процесс отладки часто требует именно контроля того, как программа отреагирует на поступающие сигналы. Если входных сигналов немного и алгоритм их поступления достаточно прост, то их

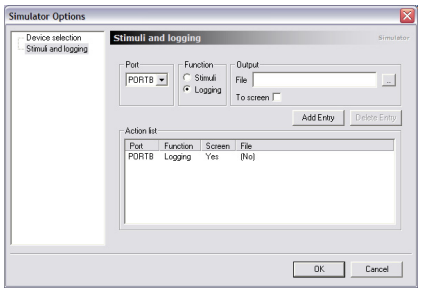
вполне можно проимитировать, устанавливая нужные значения в нужные моменты времени непосредственно в соответствующих битах регистров PINx, как было сказано ранее. Но этот способ сильно усложняется, если число входных сигналов растет, и становится почти невозможным, если частота их поступления высока.

Решение этой проблемы заключается в использовании так называемой «*стимуляции*» портов микроконтроллера, т.е. имитации поступления на них внешних сигналов. Реализуется это при помощи заранее подготовленного текстового файла с расширением «**sti**», в котором последовательно перечислены условные моменты (в машинных тактах работы микроконтроллера), когда состояние сигналов на порте меняется, и, разумеется, сами эти значения сигналов. То есть файл стимуляции имеет примерно следующее содержимое:

```
000000000:00
000000009:AB
000000014:AC
999999999:FF
```

На рисунке показано, что в начальный момент все сигналы, подаваемые на порт, имеют низкий логический уровень (00). В момент наступления 9-го машинного такта состояние сигналов меняется на 0xAB, а к 14-ому такту на 0xAC и т.д. Количество строк в файле ограничено значением 999999999.

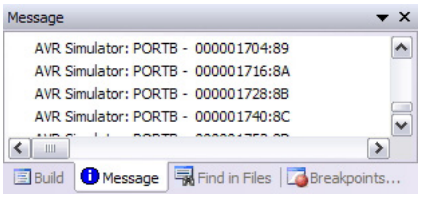
Для стимуляции используется отдельный режим отладки, настраиваемый группой параметров **Stimuli and logging** ранее рассматриваемого окна настроек эмулятора.



Вы должны указать порт, который будет подвержен стимуляции, выбрав его из списка **Port**, а так же задать файл со стимулирующей последовательностью **Input**, после чего нажать кнопку **Add Entry** для добавления заданной стимуляции к списку операций **Action List**. Для каждого порта вы можете указать свой файл аналогичным способом, после чего нужные уровни поступят в нужные моменты времени на соответствующие «выводы портов» автоматически во время отладки, вам останется лишь следить за реакцией на это вашей программы.

Кроме стимулирования существует обратная задача – *протоколирование* сигналов, формируемых микроконтроллером – **Logging**. В этом случае сигналы выбранного порта сохраняются в файл с расширением «**log**» точно в том же виде, как и при стимуляции. Выбор функции – стимуляция или протоколирование – осуществляется выбором соответствующей опции **Function**.

Если выбрано протоколирование порта, то файл можно и не задавать, если активировать опцию **To screen**. В этом случае по мере смены уровней на выбранном порту в окне **Message** будут выводиться соответствующие сообщения такого вида:



Вывод этих сообщений не зависит от того, ведется ли запись протокола в файл или нет.

Создание файлов стимуляции – довольно-таки утомительная процедура, особенно для длительных и сложных последовательностей. Облегчить ее можно, если использовать дополнительные утилиты сторонних разработчиков – см. главу «Дополнительные средства».

СРЕДСТВА ПОДДЕРЖКИ АППАРАТНОЙ ОТЛАДКИ

Под аппаратной отладкой понимаются средства, подключаемые к компьютеру через один из имеющихся интерфейсов, и обеспечивающих исполнение программы в реальном микроконтроллере, но под контролем среды AVR Studio.

Это позволяет полностью исключить несоответствия программной эмуляции микроконтроллера. Яркий пример – рассмотренный способ стимуляции портов. Очевидно, что такая стимуляция осуществляется строго синхронно с работой микроконтроллера, т.к. моменты изменения уровней сигналов привязаны к числу машинных тактов. В реальных схемах моменты поступления внешних сигналов никак не связаны с работой микроконтроллера, и с точки зрения программы являются практически случайными. Кроме того, программная эмуляция попросту невозможна для ситуаций с большим периодом повторения – уже отладка процессов, длящихся десятки секунд, в режиме эмуляции становится утомительно долгой, что тогда говорить о процессах, длящихся минуты и часы!

Так же практически невозможна эмуляция различных сложных и быстродействующих интерфейсов, например, CAN или USB, аналоговые устройства так же не эмулируются в принципе.

Обзор средств

Большинство средств аппаратной отладки разработано и поставляется на рынок самой фирмой Atmel, имеющиеся на рынке образцы сторонних производителей – лишь упрощенные версии фирменных устройств, либо их функциональные аналоги.

Аппаратная отладка может осуществляться либо по стандартному интерфейсу JTAG, встроенному в некоторые типы микроконтроллеров, либо по интерфейсу Debug Wire, так же присутствующему во многих моделях микроконтроллеров. Необходимость аппаратной поддержки указанных интерфейсов накладывает ограничения на применимость средств – многие микроконтроллеры принципиально не могут работать с этими средствами.

Имеются средства, которые облегчают процесс отладки, не заменяя эмуляцию, но дополняя ее – однако, это в сущности лишь макетные платы с готовым интерфейсом «визуализации» данных – либо в виде различных дисплеев, либо в виде возможности вывода информации из микроконтроллера в компьютер и отображения ее в окне терминальной программы.

Все фирменные средства отладки (и, в том числе, программирования) микроконтроллеров приведены в справочном файле, открываемом по команде меню «Help» AVR Tools User Guide. В этом же файле приведены подробные инструкции по их использованию. В рамках данной статьи рассмотреть все средства достаточно подробно невозможно, поэтому ограничимся лишь их кратким перечнем с указанием основных особенностей.

ICE50 и **ICE40** – эмуляторы-отладчики, поддерживают почти все микроконтроллеры, обеспечивают полный функционал отладки, включая все виды точек останова, поддержку аналоговой периферии, сторожевого таймера, режимов «сна» микроконтроллера и т.п. Отличаются комплектностью и количеством поддерживаемых микроконтроллеров. Подключаются вместо реального микроконтроллера в схему пользователя и имитируют его работу.

JTAGICE – эмулятор-отладчик, поддерживающий только микроконтроллеры со встроенным интерфейсом JTAG. Дополнительно обеспечивает возможность программирования микроконтроллеров. В отличие от **ICE50**, не эмулирует работу микроконтроллера, а подключается к имеющемуся микроконтроллеру на плате пользователя, т.е. обеспечивает наблюдение за работой конкретного экземпляра контроллера.

ICE200 – несколько упрощенная версия **ICE50** со слегка усеченным функционалом.

AVR Dragon – отладочный комплекс в виде платы, на которой предусмотрена зона макетирования, т.е. в некоторых случаях непосредственно на плате этого устройства пользователь может собирать свои схемы. Обеспечивает поддержку отладки как по интерфейсу JTAG, так и Debug Wire, реализует все режимы программирования микроконтроллеров. Поддерживает все микроконтроллеры (часть – при помощи дополнительных средств).

Особенности использования

Как было сказано, аппаратные отладочные средства делятся на 2 типа: эмулирующие микроконтроллер и наблюдающие за микроконтроллером. Первый тип, не смотря на гибкость и широкий спектр поддержки контроллеров, не может быть на 100% полным аналогом, т.е. все равно в силу своей работы может иметь отклонения от поведения реальных кристаллов. Большинство таких отклонений известны и перечислены в соответствующих фирменных

документах, однако полной гарантии в отсутствии новых нет.

С другой стороны, «наблюдающие» через JTAG или Debug Wire отладчики категорически не подходят для отладки многих моделей контроллеров…

Наконец, всем типам аппаратных средств все равно присуща одна главная проблема: допуская в любой момент приостановку исполнения программы, они тем самым нарушают «реальность» окружения микроконтроллера. Скажем, остановив контроллер, они не останавливают сигналы с датчиков, подключенных к нему. В этом случае программа исполняется все равно не в той среде, как при реальной работе устройства – и это следует учитывать при отладке.

ПРОЦЕСС ОТЛАДКИ ПРОГРАММЫ

Итак, рассмотрены практически все средства обеспечения отладки – от окна AVR Studio до внешних аппаратных отладчиков. Настала пора рассмотреть в деталях сам процесс отладки, т.е. как используются и взаимодействуют все рассмотренные средства.

Начинается процесс отладки с нажатия кнопки  или соответствующей горячей комбинации клавиш **Ctrl-Shift-Alt-F5**. Рабочее пространство AVR Studio при этом видоизменяется, подготавливаясь к процессу отладки. Если используются средства аппаратной поддержки – они инициализируются (их подключение должно быть сделано ранее). В окне с исходным текстом появляется желтая стрелка, указывающая на строку программы, готовую к исполнению:

```
#include <avr/io.h>
// пример программы
volatile unsigned char tmp=100;

int main(void){
    DDRB = 255;
    while (tmp){
        PORTB++;
    }
}
```

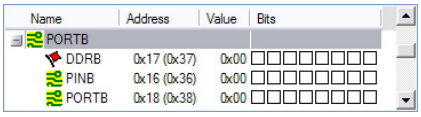
Теперь, в зависимости от намерений программиста, можно открыть любое из рассмотренных ранее окон для просмотра переменных, памяти, регистров и т.п. – на этом подготовительные операции завершены.

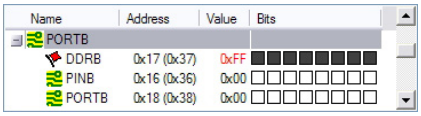
Пошаговое исполнение программы

После подготовки начинается, собственно, процедура отладки. Обычно она заключается в пошаговом исполнении программы, т.е. исполнению операторов одного за другим. Под пошаговым исполнением подразумевается то, что каждый оператор исполняется только после того, как программист даст на это команду – нажмет кнопку  или  (т.е. команды Step Into или Step Over – см. меню Debug – отладка). Содержимое окна немного изменится:

```
#include <avr/io.h>
// пример программы
volatile unsigned char tmp=100;

int main(void){
    DDRB = 255;
    while (tmp){
        PORTB++;
    }
}
```

Как видите, указатель передвинулся на очередную строку программы (а сама строка выделилась). Дальнейшее нажатие клавиш F10 или F11 позволит последовательно исполнить и остальные операторы программы, наблюдая по ходу дела за изменениями, осуществляемыми ими над переменными. На рисунке показано содержимое регистра DDRB до исполнения оператора DDRB=255 и после:



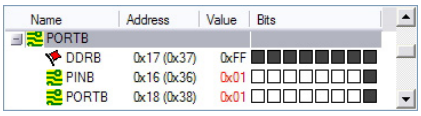


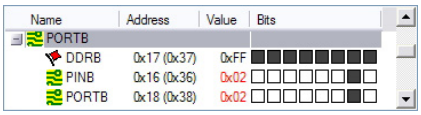
А окно программы при этом будет уже таким:

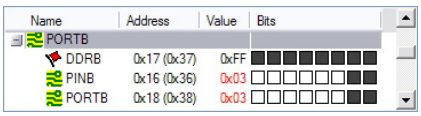
```
#include <avr/io.h>
// пример программы
volatile unsigned char tmp=100;

int main(void){
    DDRB = 255;
    while (tmp){
        PORTB++;
    }
}
```

Далее в программе следует оператор бесконечного цикла, в котором постоянно увеличивается на 1 содержимое PORTB, т.е. на выводах порта формируется возрастающая двоичная последовательность сигналов:

Окно периферии







Пошаговое исполнение может осуществляться и в том случае, когда открыто окно дизассемблера – в этом случае каждый шаг будет соответствовать одной ассемблерной команде.

Следует отметить, что нормальная отладка возможна только при компиляции программы с отключенной оптимизацией (см. главу «Параметры компиляции проекта»). При включении оптимизации при отладке могут наблюдаться «чудеса»: то порядок исполнения строк программы не соответствует ожиданиям, то некоторые переменные недоступны для наблюдения в окне Watch, или же в какой-то строке программы невозможно поставить точку останова. Эти эффекты – следствие работы оптимизатора, который просто может выбросить за ненадобностью некоторые строки программы, изменить (не нарушая логику работы) последовательность выполнения операторов или удалить ненужные куски кода вообще.

К сожалению, без оптимизации размер кода получается существенно больше, чем с оптимизацией, и для микроконтроллеров с малым объемом памяти никакой отладки вообще может не получиться. Тут придется идти на компромисс: либо отлаживать программу с «чудесами», стараясь уследить за тем, что она делает, либо собрать проект без оптимизации и отладить его на микроконтроллере, максимально близком к нужному, но с бОльшей памятью – так как многие контроллеры обладают сходной периферией, то погрешность такого метода минимальна.

Автоматическое исполнение программы

Кроме исполнения программы по шагам под контролем пользователя, имеются и режимы автоматического исполнения – как по шагам, так и в непрерывном режиме.

Автоматическое исполнение по шагам заключается в том, что AVR Studio самостоятельно подает сама себе команды **Step Into**. Этот режим позволяет понаблюдать, как программа исполняется – после каждого автоматического шага обновляются значения во всех окнах, и программист, наблюдая за этим процессом, может сделать какие-то выводы.

Запускается автовыполнение по шагам кнопкой  (**Auto Step**). Остановка этого процесса осуществляется командой **Break** (кнопка ).

Необходимость обновления большого количества информации на дисплее делает этот режим достаточно медленным. Если необходимо отладить программу, содержащую большие участки уже проверенного кода или же длительные циклы, можно воспользоваться режимом автоматического исполнения, который включается командой **Run** (кнопка). В этом случае вся информация во всех окнах и панелях «замораживается», в то время как программа «исполняется» на полной скорости. В случае использования аппаратных отладчиков происходит действительное исполнение программы, т.е. микроконтроллер работает на заданной тактовой частоте, а в случае эмуляции – «виртуальное» исполнение происходит на максимально возможной скорости эмуляции, обеспечиваемой мощностью компьютера.

Остановить режим исполнения так же можно командой **Break**.

Точки останова

Кроме принудительной остановки исполнения программы, в котором не очень много пользы, имеется гораздо более удобный способ – указание точки останова (**breakpoint**).

AVR Studio реализует 2 типа точек останова – программная (**Program breakpoint**) или по изменению данных (**Data breakpoint**). Программная точка останова просто помечает строку программы, дойдя до которой процесс автоисполнения будет остановлен,

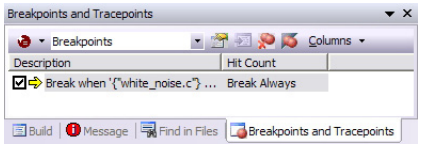
при этом сама строка еще не будет выполнена. Такие точки останова очень полезны при отладке долгих процессов, прерываний и т.п. В этом случае весь неинтересный для программиста код исполняется автоматически и достаточно быстро, а со строки, отмеченной точкой останова, отладка ведется по шагам.

Остановка по изменению данных происходит лишь в том случае, когда программа изменит значение указанной переменной (или указанной области памяти). Этот режим очень полезен для поиска мест в программе, где происходит незапланированное изменение переменной. Например, в ходе отладки обнаруживается, что глобальная переменная **tmp** принимает значение, которое программист не предусматривал, в результате чего программа исполняется неверно. Если программа состоит из нескольких модулей, а каждый модуль – из сотен строк, то простым анализом исходного текста программы найти место этого изменения очень сложно, а если это связано с переменными-указателями, то может и вообще невозможно. В этом случае программист задает точку остановки по любому изменению переменной **tmp** и запускает программу на исполнение. Всякий раз, как только произойдет модификация содержимого переменной, автоисполнение будет прекращено на первом же операторе после модификации. Проанализировав это место в тексте программы и сопоставив при необходимости его с текущим содержимым других переменных, программист либо продолжает автоисполнение (если это место вне подозрений), либо приступает к исправлению найденной ошибки.

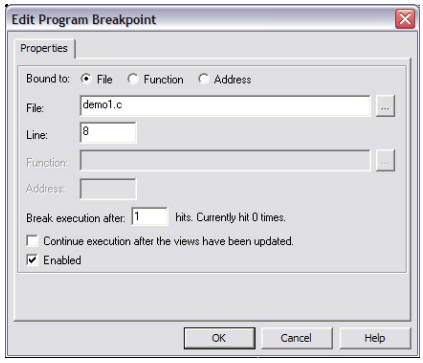
Установка обычных точек остановки происходит простым нажатием кнопки  (или командой **Toggle Breakpoint**), при этом строка, в которой находится курсор (текстовый, а не «мышиный»), отмечается красной точкой:

```
#include <avr/io.h>
// пример программы
volatile unsigned char tmp=100;

int main(void){
    DDRB = 255;
    while (tmp){
        PORTB++;
    }
}
```

Одновременно с этим в окне **Breakpoints and Tracepoints** появляется новая строка:



В этом окне указано, что точка остановки установлена в модуле **demo1.c** в строке с номером **8**, эта точка в настоящее время активирована (отмечена галочкой) и вызывает остановку всегда. Данное окно позволяет гибко управлять точками останова. Непосредственно в нем можно удалить точку, выделив строку и нажав ***Del***, или временно деактивировать, «сняв» галочку с нужной точки (в этом случае останова не будет происходить, хотя сама точка останется). Но гораздо больше возможностей предоставляет кнопка  в этом окне (или команда **Properties** из всплывающего меню), которая открывает следующее окно:



В этом окне можно настроить много параметров установленной программной точки останова. Начнем рассмотрение снизу вверх, т.к. внизу расположены общие для разных режимов опции. Во-первых, опция **Enabled** управляет активностью точки (галочка в списке). Во-вторых, имеется возможность не останавливать исполнение при проходе точки, а только обновить содержимое всех окон AVR Studio – за это отвечает опция **Continue execution after the views have been updated**. В-третьих, имеется возможность остановиться не сразу, а лишь после определенного количества проходов по точке (очень удобно при отладке циклов) – для этого следует указать в окне **Break execution after** значение, большее 1. Рядом с этим окном приводится для справки число проходов через точку к текущему моменту (на рисунке – 0 проходов).

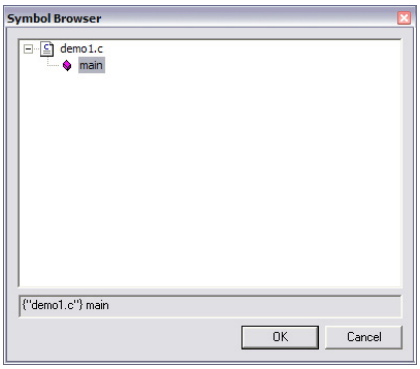
Теперь рассмотрим остальные опции сверху вниз.

Bound to – указывает, к чему применяется точка. Варианты возможны такие: **File** – строка в файле модуля, **Function** – функция в модуле или **Address** – адрес машинного кода в программе. Переключения области точки изменяет вид остальной части окна, делая одни опции активными, а другие – недоступными.

Если точка назначается строке в файле, то вы можете указать собственно имя файла в поле **File** и номер строки в нем – поле **Line** (но гораздо проще это сделать, как было сказано ранее – командой **Toggle Breakpoint**).

Для точки на функции станет активным поле выбора функций программы – **Function**. Вы должны будете либо ввести имя функции в этом поле (в особом формате), либо, что удобнее, нажать кнопочку рядом с полем и выбрать функцию из списка (*см. рисунок в следующей колонке*).

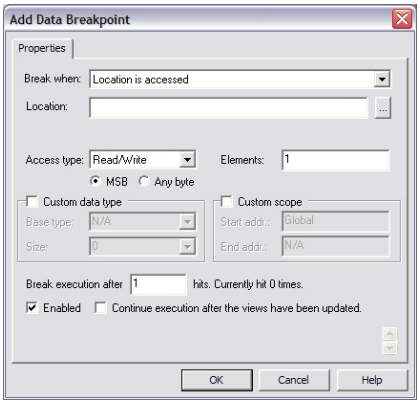
В этом окне в виде древовидной структуры показаны все функции проекта, достаточно выбрать нужную и нажать кнопку **OK**.



Наконец, если выбран конкретный адрес – активируется поле для его ввода. Следует учесть, что если окажется, что заданный адрес находится «внутри» какого-то оператора, то в окне текста программы никакой отметки соответствующей строки не будет, увидеть ее можно будет лишь в окне дизассемблера, однако остановка будет происходить все равно. Только вот с выделением строки, на которой произошла остановка, может возникнуть небольшая проблема: после оптимизации, как было сказано, не всегда имеется возможность однозначно определить, какому именно оператору Си соответствует конкретная ассемблерная команда. В этом случае после остановки может быть выделена строка с оператором, следующим за тем, внутри которого произошла остановка.

Возможностей программных точек останова, как видите, достаточно немало, но возможностей остановки по изменению данных существенно больше!

Установить точку остановки по изменению данных можно либо при помощи меню «**Debug**» (команда **New Breakpoint – Data breakpoint**), либо непосредственно в окне просмотра точек остановки **Breakpoints and Tracepoints**. Во втором случае для этого следует использовать кнопку  или команду **New** из всплывающего меню. При любом способе открывается окно настройки параметров точки останова:



Параметров, как видите, существенно больше, чем для программной точки. Самые нижние опции – точно такие же, как и ранее, а вот верхние следует рассмотреть подробно.

Самый верхний элемент – список условий срабатывания точки останова – **Break when** (остановить, когда). В раскрывающемся списке перечислены все возможные варианты условий:

· **Location is accessed** – осуществлен любой доступ к переменной

· **Location content is equal to a value** – значение переменной совпадает с указанным значением

· **Location content is not equal to a value** – значение переменной не равно указанному значению

· **Location content is greater than a value** – значение переменной больше указанного значения

· **Location content is less than a value** – значение переменной меньше указанного значения

· **Location content is greater than or equal to a value** – значение переменной больше или равно указанному значению

· **Location content is less than or equal to a value** – значение переменной меньше или равно указанному значению

· **Location content is within a range** – значение переменной находится в указанном диапазоне

· **Location content is outside a range** – значение переменной вне указанного диапазона

· **Bits of a location is equal to a value** – определенные биты в переменной имеют заданные значения

· **Bits of a location is not equal to a value** – определенные биты в переменной не совпадают с заданным значением

Если для указанной переменной (в поле **Location**) выполняется выбранное условие – происходит срабатывание точки и автовыполнение программы останавливается (разумеется, лишь в том случае, если все другие условия этому не противоречат). В зависимости от того, какое именно условие выбрано, меняются остальные поля ввода значений. Так, например, для проверки битов в переменной, появляется поле ввода маски **Bitmask** (в котором надо отметить единичными значениями те биты, которые анализируются в переменной), а при проверке значения на попадание в диапазон – появляются поля ввода минимального и максимального значения. **Value** – это поле значения, с которым сравнивается указанная в **Location** переменная. Выбор переменной проще всего осуществить из списка, открывающегося по нажатию кнопки с многоточием рядом с полем **Location** – вид этого списка совпадает с ранее рассмотренным списком выбора функций, с той лишь разницей, что выбирать следует переменные (локальные или глобальные).

Далее следуют поля и опции «тонкой» настройки режима контроля значения переменных.

Поле **Access type** позволяет указать способ обращения к переменной, после которого осуществляется проверка условия. Есть три варианта: **Read/Write** (любое обращение к переменной), **Read only** (только чтение) и **Write only** (только запись). В первом случае проверка происходит после любого обращения к переменной, во втором – только после считывания, в третьем – только после записи.

Далее следует опция выбора способа контроля многобайтных переменных: **MSB** – только старший байт или **Any byte** – любой байт.

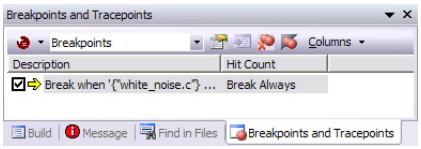
Группа опций **Custom data type** (пользовательский тип данных) позволяет задать режим проверки переменных нестандартных типов. Активировав эту опцию, следует выбрать базовый тип переменной **Base type** и указать ее фактический размер в байтах **Size**. Это необходимо делать в тех случаях, если контролируются переменные не стандартных типов, а введенных пользователем.

Группа опций **Custom scope** (область просмотра) позволяет указать область контролируемой памяти, задав начальный и конечный адреса – поля **Start addr** и **End addr** соответственно.

Следует отметить, что в большинстве случаев пользователю не нужно производить настройку режимов контроля переменных, согласившись с параметрами по умолчанию.

Возвращаясь немного назад, к окну просмотра содержимого памяти, следует сказать, что в его всплывающем меню имеется команда установки точки останова по изменению любой области памяти без привязки к конкретной переменной **Add Data breakpoint**. Пользоваться этой возможностью очень удобно, например, для контроля вершины стека программы.

Когда происходит срабатывание точки остановки, она отмечается стрелочкой желтого цвета:



Следует дополнительно отметить, что в окне **Breakpoints and Tracepoints** могут быть указаны (а в тексте программы – расставлены) не только точки остановки, но и *точки трассировки* (**Tracepoints**), для которых имеются соответствующие команды. Точка трассировки позволяет проследить момент «прохода» программы через указанную строку, т.е. позволяет получить в некотором смысле «протокол» исполнения программы. Однако эта возможность реализуется только при наличии средств аппаратной поддержки, в режиме «виртуальной» отладки недоступна и потому не рассматривается.

Альтернативные средства отладки

Не смотря на достаточно широкие возможности виртуального исполнения программы, часть проблем с их помощью решить невозможно без использования либо аппаратных средств, либо без натурных испытаний устройства.

Частично эту проблему можно решить при помощи других средств отладки, наиболее интересным из которых следует признать программы симуляции электронных схем. К сожалению, все эти программы – исключительно коммерческие, т.е. далеко не бесплатные.

Одна из очень удачных программ для этого – небезызвестный **ISIS Proteus Professional** фирмы **Labcenter Electronics**. Эта программа позволяет «нарисовать» принципиальную схему устройства с микроконтроллером, используя «интерактивные» элементы, а затем «загрузить» в микроконтроллер написанную программу и «подать питание» на схему. При этом программно моделируется поведение всех элементов – от микроконтроллера до резистора и транзистора, по возможности все происходящие в схеме изменения отображаются практически в реальном времени на дисплее – «загораются» светодиоды, «вращаются» моторы, динамики издают звуки и т.п. Кроме чисто визуальных средств можно использовать «виртуальные» инструменты – осциллограф, генератор сигналов и т.п.

Эта программа позволяет не только увидеть внешние эффекты, демонстрирующие результат работы программы, но и так же, как и в AVR Studio, проводить отладку по шагам, просматривая содержимое переменных и памяти. К сожалению, изменить вручную содержимое переменных в этом случае невозможно.

Начиная с версии AVR Studio 4.16 появилась возможность интеграции с установленным Proteus ISIS. Если протеус был уже установлен к моменту установки AVR Studio, то в списке платформ для отладки (см. главу «Мастер проектов» – раздел о выборе платформы отладки **Select debug platform and device**) появится платформа Proteus VSM Viewer. Если выбрать эту платформу, то при запуске отладки произойдет «внедрение» протеуса внутрь окна AVR Studio – при этом схему отлаживаемого устройства можно создать там, а отлаживать в студии. У этого гибрида масса достоинств – все плюсы отладки по точкам останова и просмотру/изменению переменных от студии и все плюсы точной имитации аналоговой периферии и схемы от протеуса. Недостаток только один – требования к памяти и мощности процессора компьютера. Так же порой эта связка может «упасть», т.е. обе программы завершаются по ошибке (редкое явление).

Proteus был бы исключительно незаменимым средством для любого разработчика, если бы не был весьма дорогостоящим продуктом.

Продолжение в №8/2010