Особенности отладки программ для микроконтроллеров семейства 8051 в среде Keil uVision

Андрей Сошкин, Владимир Трубчанинов (Москва)

В статье рассматриваются специальные приёмы отладки программ для микроконтроллеров 8051. На примерах рассмотрена программная имитация внешних воздействий.

Среда Keil – пожалуй, лучшая среда разработки для процессоров с архитектурой 8051. Это касается не только чистоты генерируемого кода, продуманности пользовательского интерфейса, удобства подключения сторонних программ, но и, самое главное, развитых механизмов отладки и эмуляции процессоров различных производителей и с различной периферией.

В дополнение к обычным возможностям отладчика, таким как установка контрольных точек, пошаговое выполнение и просмотр текущего значения оперативной памяти и регистров, отладчик позволяет имитировать различные внешние воздействия, а также обладает возможностью написания собственных процедур и функций имитации внешних воздействий на языке Си.

В статье последовательно освещены следующие вопросы:

- обзор общих возможностей отладчика;
- варианты использования контрольных точек;
- имитация внешних воздействий на отлаживаемую программу;
- организация собственных процедур и функций имитации.

Для обеспечения скорейшего практического применения дополнительных возможностей отладки в интегрированной среде Keil рассматриваются несложные программы на языке Си.

Чтобы запустить режим отладки в среде Keil, необходимо откомпилировать проект и запустить собственно режим отладки одним из трёх способов:

- нажать комбинацию клавиш Ctrl+F5;
- выбрать пункт меню *Debug* >> *Start\Stop Debug Session*;
- нажать кнопку *Start\Stop Debug Session* на панели инструментов.

Обзор возможностей отладчика

В режиме отладки окно программы примет вид, показанный на рисунке 1. Область 1 соответствует появляющимся кнопкам панели управления отладчиком. Область 2 отображает состояние регистров процессора. Область 3 отображает инструкции процессора на языке ассемблера с привязкой к исходному коду на языке Си или на ассемблере и соответствует специальному окну Disassembly. Область 4 соответствует окну истории команд пользователя и сообщений отладчика. Область 5 соответствует окну просмотра состояния переменных программы. Область 6 соответствует строке ввода текстовых команд отладчика.

Для понимания возможностей управления отладчиком рассмотрим кнопки панели управления:

- сброс стояния процессора (*Reset CPU*);
- запуск программы на непрерывное выполнение (*Run*);
- завершение выполнения программы (*Halt*);
- пошаговое выполнение программы (*Step into*);
- пошаговое выполнение без входа в вызываемые подпрограммы (*Step over*);
- автоматическое выполнение до выхода из подпрограммы (*Step out*);
- автоматическое выполнение до выделенной курсором команды (*Run to cursor*);
- отобразить текущую позицию счётчика программы;
- разрешение/запрет режима трассировки;
- отобразить результаты трассировки;
- вызов окна дизассемблера;
- вызов окна просмотра переменных и стека;
- вызов статистики покрытия кода по блокам программы;
- вызов окна просмотра передачи данных через последовательный порт;
- вызов окна просмотра области памяти;
- вызов окна статистики выполнения по блокам программы;
- вызов окна логического анализатора;
- вызов панели кнопок пользователя.

Окно регистров (область 2) отображает текущее состояние регистров процессора. Окно просмотра переменных (область 5) организовано в виде четырёх страниц. Страница Locals отображает локальные переменные текущей выполняемой подпрограммы (если программа написана на языке Си), две следующие страницы служат для отображения любых переменных, заданных пользователем, последняя страница *Call Stack* отображает последовательность вызова вложенных подпрограмм.

Окно отображения содержимого оперативной памяти позволяет не только просматривать, но и изменять значения отдельных ячеек памяти. Для удобства анализа информации окно содержит четыре страницы. Начальный адрес отображения данных на странице определяется выражением, которое может быть как именем переменной из текста программы, так и числовым значением.

В системе Keil существует определённое правило для ввода и вывода адресных значений: они начинается префиксом, за которым следует числовое значение адреса. Префиксы адресных значений определяют тип памяти и имеют приведённые в таблице 1 обозначения.

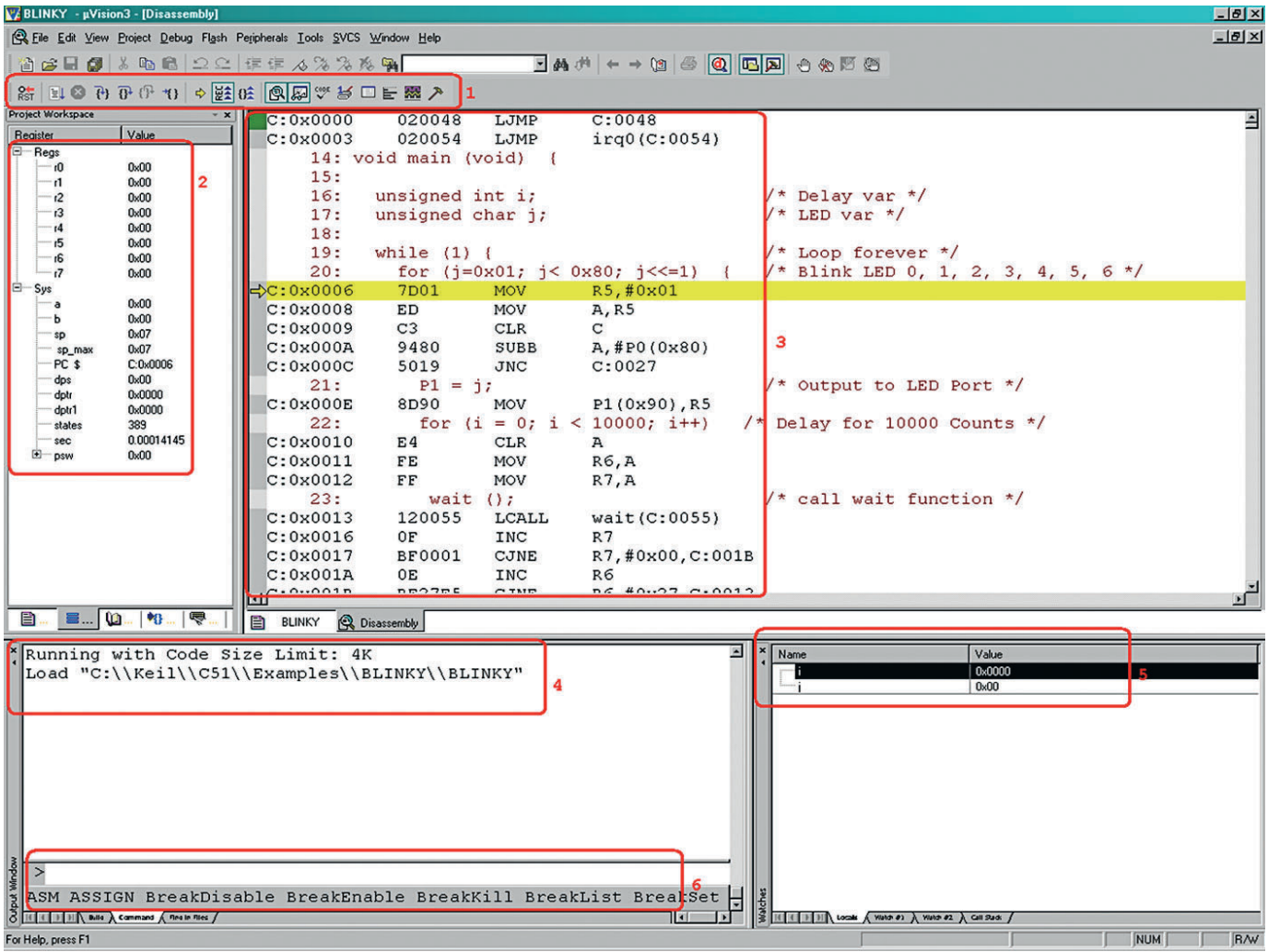

Рис. 1. Окно программы в режиме отладки

Страничная организация окна просмотра позволяет сгруппировать переменные в логически связанные множества и тем самым облегчить визуальное восприятие информации.

Для добавления переменных в окно просмотра существует ряд удобных способов:

- выделить название переменной в исходном тексте, установив курсор рядом с именем, щелчком правой кнопки мыши вызвать контекстное меню, выбрать пункт *Add to Watch window*;
- выбрать одну из страниц отображения переменных, нажать клавишу F2 и ввести символьное название переменной;
- добавить переменную с помощью текстовой команды отладчика.

Формат вывода значений переменных может быть установлен с помощью контекстного меню или текстовой команды отладчика. Удалить переменную из списка просмотра можно нажатием на клавишу Delete.

Следует отметить, что значения, отображаемые окнами просмотра переменных или регистров, актуальны на момент достижения программой активной точки останова. Для отображения актуального содержимого регистров и переменных в процессе выполнения программы следует установить соответствующий режим *Periodic Window Update* в главном меню программы.

Использование контрольных точек для отладки программ

Наиболее часто применяемым приёмом при отладке программы является расстановка точек останова в программе. Отладчик среды Keil поддерживает развитую логику остановки программы, что может значительно ускорить нахождение ошибок.

В первую очередь, точки останова характеризуются типом:

- тип E подразумевает остановку выполнения программы перед началом выполнения процессором инструкции с заданным адресом;
- тип A подразумевает остановку выполнения программы перед доступом (чтение или запись) к оперативной памяти;
- тип C подразумевает остановку выполнения программы при условии истинности какого-либо условного выражения.

Точки останова типа E используются для разделения непрерывного

Таблица 1. Префиксы адресных значений

Префикс	Область	Описание
B:	BIT	Побитно адресуемая внутренняя оперативная память процессора
C:	CODE	Память хранения программы
Bx:	CODE BANK	Банк внешней памяти хранения программы
D:	DATA	Внутренняя оперативная память процессора
I:	IDATA	Косвенно адресуемая внутренняя оперативная память процессора
X:	XDATA	Внешняя оперативная память

выполнения программы на этапы. После остановки программы можно детально проанализировать состояние переменных и регистров процессора. Таким образом, реализуется проверка входных и выходных данных для любого из блоков программы.

После нахождения блока программы с несоответствующими результатами можно повторно запустить выполнение блока в пошаговом режиме, точно определить место в программе, формирующее неправильный результат, и локализовать ошибку.

Другим методом поиска ошибок в программе является определение момента присваивания недопустимого значения переменной. *Точка останова типа C* реализует механизм контроля одной или нескольких переменных и немедленно останавливает выполнение программы при присваивании переменной недопустимых значений. Программист может детально проанализировать выделенную последовательность команд, приводящую к появлению ошибки.

Следует понимать, что проверка условия осуществляется после выполнения каждой инструкции процессора, что сильно замедляет общее быстродействие системы, поэтому большое количество проверок задавать нецелесообразно.

Точка останова типа A помогает выполнять системную отладку без привязки к коду программы. Дело в том, что в сложных микропроцессорных системах часто реализуются механизмы прямого доступа устройств к оперативной памяти или отображения управляющих регистров устройств в оперативную память процессора. Изменение состояния этих устройств и оперативной памяти может происходить асинхронно с основным процессом выполнения программы. Точки останова типа A позволяют отслеживать доступ (чтение или запись) к ячейкам оперативной памяти и регистрам процессора и тем самым наблюдать за изменением состояния устройств.

Возможна также ситуация, когда программа производит обработку или пересылку массивов данных, хранящихся в оперативной памяти. Точка останова на доступ к оперативной памяти позволяет отслеживать и такие события.

Одной из сложных (для поиска ошибок) является ситуация некорректного доступа к оперативной памяти, когда хранящиеся в ней данные затираются другими данными, осуществляя перекрытие областей памяти. Отследить момент некорректной записи данных можно с помощью точки останова типа A.

Ещё один схожий класс точек останова – это прерывание по критической ошибке доступа к отсутствующей области оперативной памяти. Причиной такой ситуации часто является ошибка адресной арифметики. Эту ситуацию можно отследить, задав диапазон действующих адресов оперативной памяти, после чего отладчик будет автоматически контролировать правильность адресов обращения к оперативной памяти.

Таблица 2. Команды управления памятью

ASM	Добавить ассемблерные инструкции к тексту программы
DEFINE	Определить символ для использования в языке сценариев отладчика
DISPLAY	Отобразить содержимое памяти
ENTER	Записать число в память
EVALUATE	Вычислить выражение и вывести результат
MAP	Определить параметры доступа для области памяти
UNASSEMBLE	Дизассемблировать код с указанного адреса

Таблица 3. Команды управления выполнением программы

Esc	Остановить выполнение программы
GO	Запустить выполнение программы в автоматическом режиме
PSTEP	Пошаговое выполнение программы без захода в подпрограммы
OSTEP	Автоматически выполнить подпрограмму
TSTEP	Пошаговое выполнение программы
UNASSEMBLE	Дизассемблировать код с указанного адреса

Таблица 4. Команды управления точками останова

BREADISABLE	Запретить одну или несколько точек останова
BREAKENABLE	Разрешить одну или несколько точек останова
BREAKKILL	Удалить одну или несколько точек останова
BREAKLIST	Вывести список точек останова
BREAKSET	Добавить точку останова в список
MAP	Определить параметры доступа для области памяти
UNASSEMBLE	Дизассемблировать код с указанного адреса

Таблица 5. Дополнительные команды

ASSIGN	Источник данных для эмулируемого последовательного порта
COVERAGE	Вывести статистику покрытия кода
DEFINE BUTTON	Создать пользовательскую кнопку на панели инструментов
DIR	Вывести список символьных имён
EXIT	Завершить отладку
INCLUDE	Считать и выполнить файл
KILL	Удалить кнопку с панели инструментов
ASSIGN	Источник данных для эмулируемого последовательного порта
COVERAGE	Вывести статистику покрытия кода
DEFINE BUTTON	Создать пользовательскую кнопку на панели инструментов
DIR	Вывести список символьных имен
EXIT	Завершить отладку
INCLUDE	Считать и выполнить файл
KILL	Удалить кнопку с панели инструментов
ASSIGN	Источник данных для эмулируемого последовательного порта

Использование команд отладчика

Кроме кнопок панели управления и команд меню, программист имеет возможность управлять отладчиком и средой выполнения с помощью текстовых команд. Во время ввода текста команды в строку синтаксический генератор отображает под строкой ввода все доступные команды и параметры в качестве подсказки. Рассмотрим команды отладчика подробнее.

Команды управления памятью приведены в таблице 2.

Команды управления выполнением программы приведены в таблице 3.

Команды управления точками останова приведены в таблице 4.

Дополнительные команды приведены в таблице 5.

Вместо ввода полного символьного представления команды иногда достаточно ввести несколько символов, которые в таблице отмечены подчёркиванием. В языке сценариев отладки и командах отладчика можно исполь-

зовать имена специализированных внутренних переменных, которые приведены в таблицах 6 и 7.

Имеется возможность интерактивно запрашивать и изменять значения программных переменных и регистров из командной строки. Для вывода значения переменной в окно сообщений достаточно ввести имя переменной. Для изменения значения переменной требуется ввести имя переменной, знак «равно» и значение переменной; как указано в таблице 8.

Панель кнопок пользователя

Очень полезным инструментом в среде Keil является панель кнопок пользователя, создаваемых с помощью команды *Define Button*. На панели кнопок, прежде всего, находится кнопка *Update Windows*, нажатие на которую приводит к обновлению значений во всех окнах просмотра.

Кнопки пользователя добавляются на панель вслед за кнопкой обновления и автоматически нумеруются. Единственное ограничение – название кнопки может содержать только латинские символы.

Для каждой кнопки может быть написан обработчик события на внутреннем языке сценариев, что позволяет программисту управлять процессом эмуляции.

Язык сценариев отладчика

В дальнейшем обзоре возможностей среды будем использовать простейшую программу, написанную на языке Си:

```
#include <REG51F.H>
void wait (void) { /* подпрог-
рамма задержки
; используется для формирования
небольших
задержек за счет вызова функции */
}
void main (void) {
unsigned int i; /* счетчик ин-
тервала задержки */
unsigned char j;/* текущее зна-
чение светящегося индикатора
выводимое в порт */
while (1) { /* основной цикл вы-
полнения программы */
for (j=0x01; j< 0x80; j<<=1) {/*
Сдвиг позиции с установленным
в единицу битом по позициям в
байте с 0 по 6 */
P1 = j; /* Вывод сформированного
байта в порт */
for (i = 0; i < 10000; i++) /*
Задержка на 10000 приращений
счетчика */
wait (); /* Вызов функции задержки*/
}
for (j=0x80; j> 0x01; j>>=1) {/*
Сдвиг позиции с установленным
в единицу битом по позициям в
байте с 6 до 0*/
P1 = j; /* Вывод сформированного
байта в порт */
for (i = 0; i < 10000; i++) /*
Задержка на 10000 приращений
счетчика */
wait (); /* Вызов функции заде-
ржки */
}
}
}
```

Приведённая программа управления выводами процессора периодически изменяет состояние порта вывода *P1* с фиксированными задержками. Файл проекта этой программы *blinky.uv3* входит в состав примеров, поставляемых со средой Keil, и хранится в папке *Examples\BLINKY* в каталоге установки среды.

Рассмотрим более подробно язык сценариев отладчика. Операторы и синтаксис языка сценариев полностью соответствует языку Си, но имеются следующие отличия:

- в языке сценариев прописные и строчные символы не различаются;
- существуют ограничения по работе с указателями и адресной арифметикой;
- язык сценариев не может использоваться для вызова процедур и функций отлаживаемой программы;
- язык сценариев не поддерживает определение структур.

Язык сценариев позволяет программисту определить поведение внешних устройств и реализовать механизмы их взаимодействия с отлаживаемой программой. Код сценариев оформляется в виде подпрограмм, которые могут вызываться программистом из командной строки или пользовательскими кнопками панели; интересной возможностью является вызов подпрограмм сценариев в качестве обработчиков точек останова.

Исходный текст подпрограмм хранится в одном или нескольких файлах, которые подключаются к проекту автоматически при запуске режима отладки или могут быть загружены в ходе отладки из командной строки.

Для автоматического подключения файла с подпрограммами сценариев в свойствах проекта следует записать название файла в строке *Initialization File* на странице *Debug* настроек проекта.

Рассмотрим возможности применения процедур сценариев. Первый пример – процедура, выводящая сообщение о состоянии нулевого бита порта ввода-вывода:

```
func void check_p10(void) {
if (PORT1 & 1) /* применяем мас-
ку для выделения состояния 0-го
бита*/
printf("Индикатор вкл!\n"); /*
```

Таблица 6. Общие встроенные переменные

$	Счётчик команд
break	Признак остановки выполнения программы
iip	Номер обрабатываемого прерывания
states	Счётчик выполненных процессором инструкций
Itrace	Признак записи файла трассировки
radix	Основание системы счисления 10 или 16

Таблица 7. Переменные эмуляции аппаратуры процессора

AINx	Выводы аналогового входа АЦП
CLOCK	Тактовая частота работы процессора
PORTx	Порт ввода/вывода общего назначения
SxIN	Буфер для ввода данных в последовательный порт
SxOUT	Буфер для вывода данных из последовательного порта
SxTIME	Признак вычисления временных задержек при последовательной передаче данных
XTAL	Тактовая частота внешнего источника, подключенного к процессору

Таблица 8. Изменение значения переменной

MDH	Вывести значение регистра MDH
R7 = 12	Присвоить регистру R7 значение 12
time.hour	Вывести значение атрибута hour структуры time
time.hour++	Увеличить на 1 значение атрибута hour структуры time
Index = 0	Присвоить переменной index значение 0

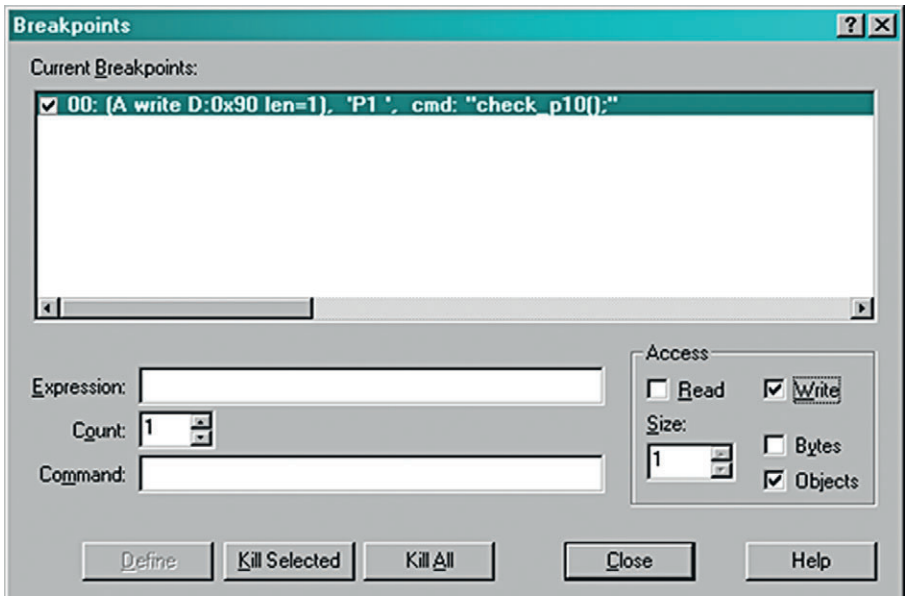

Рис. 2. Окно управления списком точек останова

```
вывод текста в окно сообщений*/
else
printf("Индикатор выкл!\n");
}
/* _ */
```

Текст процедуры набираем в редакторе среды Keil и сохраняем в файле под названием *check.ini* в каталоге текущего проекта.

Обратите внимание, что после процедуры в конце файла содержится текст пустого комментария; это – обязательный элемент, который должен добавляться в конец ini-файла для его правильной компиляции отладчиком (можно ограничиться пустой строкой). Фактически последний синтаксический элемент текста не должен находиться в последней строке файла – такова особенность встроенного компилятора языка сценариев. После сохранения файла открываем проект *blinky.uv3*, запускаем отладчик и вводим в командной строке

```
INCLUDE check.ini
```

Отладчик автоматически загрузит указанный файл и проведёт его анализ и трансляцию. Таким образом, процедура *check_p10* становится доступной для вызова. Чтобы «привязать» выполнение процедуры к событию изменения состояния порта ввода-вывода, используем точку останова по доступу (тип «A»), вводя следующую команду:

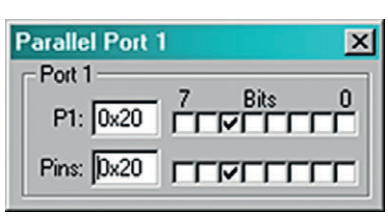

Рис. 3. Окно портов ввода-вывода

```
BS WRITE P1 , 1 , "check_p10();"
```

Теперь запустим программу на выполнение вводом команды *GO* или с помощью визуальных средств отладчика. В окне сообщений наблюдаем вывод сообщений *«Индикатор вкл!»* и *«Индикатор выкл!»*. Разумеется, сообщение об установке бита будет встречаться значительно реже сообщения о том, что бит сброшен.

Остановим выполнение программы и, нажав сочетание клавиш Ctrl+B, выведем на экран окно управления списком точек останова, показанное на рисунке 2. В этом окне строка *Expression* соответствует условию проверки. Строка *Command* соответствует команде выполнения (обработчику точки останова) на языке описания сценариев. Счётчик *Count* указывает, сколько раз должно выполняться условие для вызова обработчика точки останова. В рамке *Access* содержатся условия доступа для соответствующей точки останова.

Попробуем, повторно вызвав окно списка точек останова, ввести в строку *Expression* значение *wait*. Кнопка *Define* станет активной; нажмём на неё и тем самым добавим точку останова типа E на вызов процедуры *wait*. Запустим отладчик на непрерывное выполнение программы; через некоторое время выполнение программы остановится на вызове процедуры wait.

Теперь кнопкой мыши сбросим «галочки», стоящие слева возле каждой точки останова, – эта возможность позволяет, не удаляя определения точки останова, устанавливать ей неактивное состояние.

Введём в строке *Expression* следующее выражение:

```
PORT1==32
```

Таким образом, мы добавили точку останова типа C. Запустим программу на выполнение и убедимся, что отладчик остановит выполнение, когда в порт выведется значение 32.

Сделаем неактивными все точки останова, запустим программу и выберем пункт *Peripherals* главного меню отладчика. Этот пункт меню формируется динамически при запуске отладчика и в точности соответствует описанию процессора из настроек проекта. По умолчанию в проекте используется стандартный процессор Intel C80C51F.

Для этого процессора определена следующая периферия: контроллер прерываний (Interrupt), порты ввода-вывода (I/O Ports), последовательный порт ввода-вывода (Serial), таймеры (Timer). Выбор каждого пункта меню приводит к отображению соответствующего окна с регистрами и элементами управления.

Можно открыть окно отображения портов ввода-вывода и просмотреть состояние используемого порта 1 в процессе выполнения программы. Изображение этого окна приведено на рисунке 3.

Визуализация цифровых сигналов

Отладчик Keil обладает мощным средством визуализации цифровых сигналов – логическим анализатором (Logic Analyzer). Средства логического анализатора позволяют полностью контролировать и документировать обмен процессора с периферией, во многом заменяя аппаратную платформу. В целом логический анализатор аналогичен подобным инструментам, присутствующим в средах разработки ПЛИС, и имеет сходное назначение – максимально полное тестирование функционирования микросхемы.

Вызов логического анализатора осуществляется с помощью кнопки меню управления. В появившееся пустое окно анализатора можно добавлять отображение состояния как регистров, так

и отдельных выводов. Добавлять в окно анализатора новые сигналы можно с помощью визуальных средств и с помощью команды *LA* отладчика:

```
LA PORT1
LA PORT&1
LA PORT&2
LA PORT&4
LA PORT&8
LA PORT&16
LA PORT&32
LA PORT&64
```

Затем в окне логического анализатора нажимаем кнопку Setup или выбираем пункт основного меню *Debug* > *Logic Analyzer*. Появляется окно редактирования отображения сигналов, показанное на рисунке 4. В окне можно настроить особенности отображения временного графика сигнала. В частности, для порта 1 настроим максимальное значение (строка *Max Value*) 0x81 (129). Для остальных сигналов установим тип (строка *Display Type*) битовый.

Теперь запустим программу на выполнение, затем остановим через несколько секунд и нажмём кнопку *All* справа вверху окна. Результат показан на рисунке 5. Обратите внимание на кнопки управления масштабом (*Zoom*), находящиеся справа вверху окна. Эти кнопки позволяют выбрать степень детализации по времени. Ещё одна особенность окна – линия среза времени; на рисунке 5 она установлена в крайнем правом положении, с отметками значений сигналов, установившихся синхронно в выбранный момент времени.

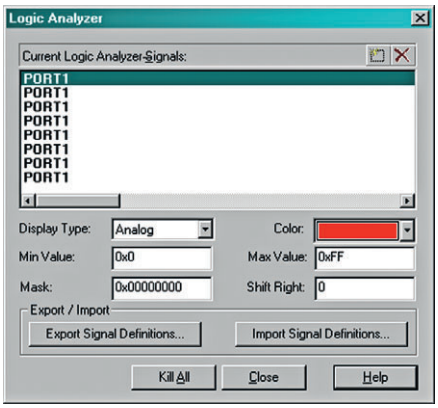

Рис. 4. Окно редактирования отображения сигналов

Организация эмуляции цифровых сигналов

Эмуляцию входных сигналов производят с помощью процедуры, имеющей в определении ключевое слово *signal*. Сгенерируем для примера на входе 2 порта 3 сигнал прямоугольной формы; заметим, что этот же вывод процессора используется для генерации прерывания IRQ0:

```
signal void irq0 (void){
while(1){
PORT3 |= 4; /* установим бит 2 */
PORT3 &= ~4; /* сбросим бит 2 */
twatch(CLOCK/4); / задержка на
1/4 секунды */
PORT3 |= 4; /* установим бит 2 */
twatch(CLOCK); /* задержка на 1
секунду*/
}
```

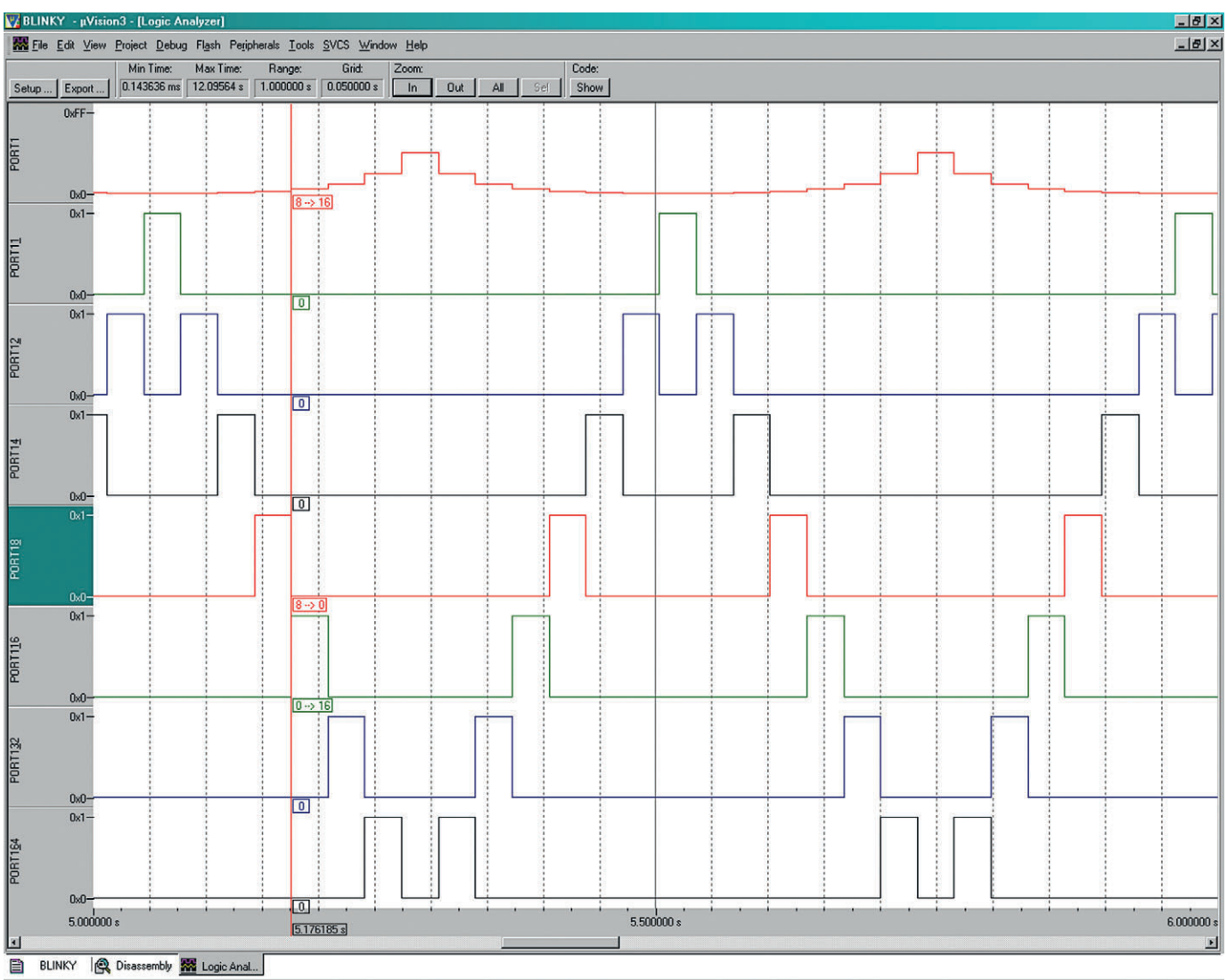

Рис. 5. Временной график сигнала

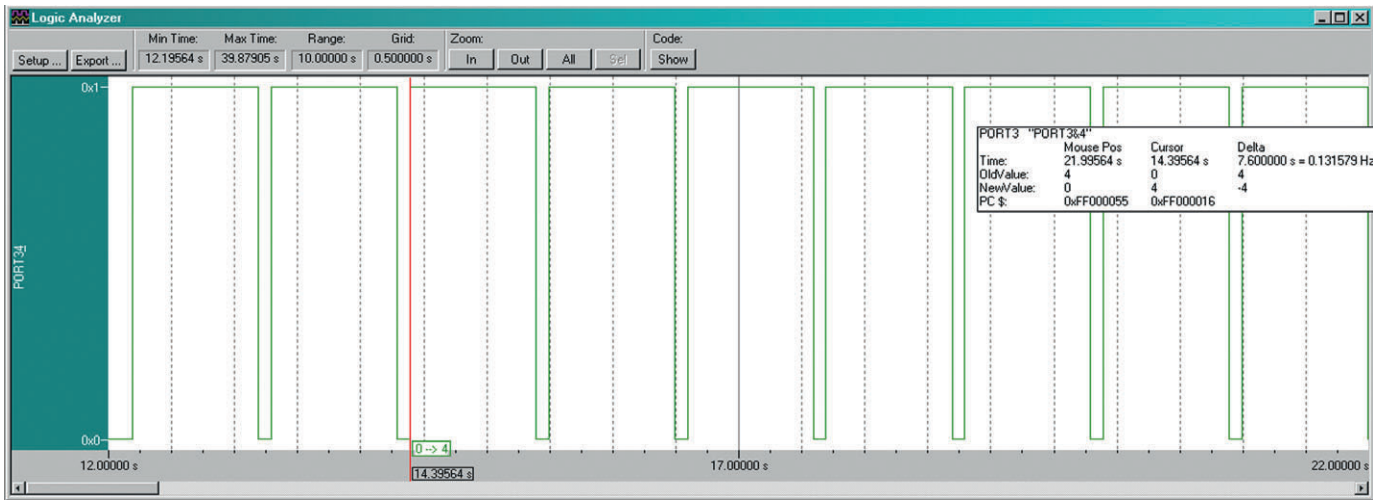

Рис. 6. Сгенерированный на выводе порта сигнал

```
}
/* */
```

Текст процедуры набираем в редакторе среды Keil и сохраняем в файле под именем *irq0.ini* в каталоге текущего проекта. Особенности процедуры:

- определение сигнальной процедуры ключевым словом *signal*;
- для формирования задержек используется встроенная функция *twatch*, которая может применяться только в сигнальных процедурах.

Функция *twatch* формирует задержки на заданное число тактов процессора; аргумент функции – целое число. В языке сценариев также имеется удобная процедура *swatch*, которая формирует задержки в секундах; аргумент функции – вещественное число, что позволяет устанавливать задержки менее одной секунды.

Настроим окно логического анализатора при помощи команд:

```
LA KILL *
LA PORT3&4
```

После сохранения файла открываем проект *blinky.uw3*, запускаем отладчик и вводим в командной строке:

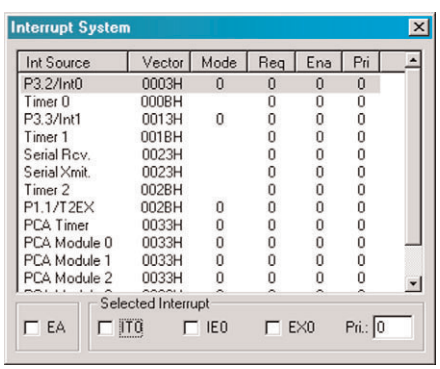

Рис. 7. Главное меню Peripherals > Interrupt

```
INCLUDE irq0.ini
irq0()
```

Таким образом, мы не только определили и загрузили в отладчик процедуру внешней генерации сигнала, но и запустили её на выполнение второй строкой.

Запускаем программу на выполнение и в окне логического анализатора видим сгенерированный на выводе 2 порта 3 прямоугольный сигнал, показанный на рисунке 6.

Остановим программу и выйдем из режима отладчика. Перед определением функции *wait* введём в программу простейший обработчик прерываний:

```
void irq0 (void) interrupt 0 {
;
}
```

Скомпилируем проект и установим точку останова на обработчике прерывания:

```
BS irq0
```

Запускаем программу в отладчике и устанавливаем генерацию сигнала, как описано выше. Откроем окно отображения состояния прерываний – главное меню *Peripherals* > *Interrupt*, как показано на рисунке 7.

Обратим внимание на периодическое мелькание «галочки» на индикаторе *IE0* и периодическое появление «1» в столбце *Req* и в строке *P3.2/Int0* таблицы. Чтобы наш обработчик сработал, вспомним про правила разрешения и маскировки прерываний и установим «галочку» сначала в индикаторе *EX0*, затем в индикаторе *EA*. Прерывание должно сработать, и выполнение программы остановится на обработчике прерывания. Отметим также продемонстрированную выше возможность изменения флагов установки, маскирования и разрешения прерываний.

Рассмотрим теперь применение отладчика для эмуляции последовательного порта ввода-вывода. Здесь, конечно, разработчики среды несколько упростили задачу и не привязали состояние соответствующих выводов процессора к выводам последовательного порта: ввод и вывод осуществляется через переменные *SIN* и *SOUT* или *SxIN* и *SxOUT*, если портов несколько, где символ *x* означает номер порта.

При написании программы работы с последовательными портами следует учитывать, что библиотечные функции ввода-вывода строк и символов в языке Си реализованы для обмена информацией через последовательный порт. Впрочем, настройка и включение последовательного порта остаётся целиком за пользовательской программой.

Для дальнейшей отладки будем использовать демонстрационную программу на языке Си, выводящую сообщение *Hello World!*:

```
#include <REG51F.H>

#include <REG52.H>
#include <stdio.h> /* использо-
вание стандартных подпрограмм
ввода/вывода*/
void main (void) {
/*******************************/
/* Настройка последовательного
порта*/
/*******************************/
SCON = 0x50; /* режим - 1, дан-
ные 8 бит, приёмник включен */
```

```
TMOD |= 0x20; /* установить 2-й
режим работы таймера 1
(8-битный счетчик с автоматичес-
кой перезагрузкой) */
TH1 = 221; /* значение счётчика
таймера 1,
для скорости 1200 бод при часто-
те тактирования 16 Гц */
TR1 = 1; /* включить таймер 1 */
TI = 1; /* установка флага TI
для активизации вывода данных */
while (1) {
P1 ^= 0x01; /* формирование пря-
моугольных импульсов
на выводе 1 порта 1 */
printf ("Hello World!\n"); /*
Вывести сообщение
в последовательный порт */
}
}
```

Файл проекта этой программы *hello.uv3* входит в состав примеров, поставляемых со средой Keil, и хранится в папке *Examples\HELLO* в каталоге установки среды.

Напишем простейшую процедуру сценария, которая понадобится нам в дальнейшем:

```
func void uart (void){
SIN = 'A';
}
/* */
```

Текст процедуры набираем в редакторе среды Keil и сохраняем в файле под именем *uart.ini* в каталоге текущего проекта. Скомпилируем программу, запустим режим отладчика и затем запустим программу на выполнение.

Если открыть окно вывода данных, передающихся через последовательный порт, то можно увидеть результаты работы программы в виде постоянно появляющихся строк сообщения. Введём следующие команды, чтобы добавить сигналы для просмотра в логическом анализаторе:

```
LA SOUT
LA (PORT1 & 0x1) >> 0
```

В окне логического анализатора настроим отображение сигналов:

- для сигнала *PORT1* установим максимальное значение отображения равным 1;
- для сигнала *SOUT* установим максимальное значение отображения равным 128.

В результате получим примерно следующие графики сигналов (см. рисунок 8). Таким образом, мы реализовали программу постоянного вывода данных через последовательный порт.

Добавим в нашу программу после команды вывода строки (в тело цикла) строку:

```
getchar();
```

Таким образом, мы изменили логику работы программы: после вывода строки символов программа ожидает ввод любого символа. После ввода символа выполняется основной цикл программы, и программа опять переходит в режим ожидания.

Добавим и настроим в окне анализатора (аналогично *SOUT*) сигнал *SIN*. Теперь выйдем из режима отладчика, скомпилируем проект и снова запустим выполнение программы в отладчике. После вывода сообщения программа ожидает ввода символа; введём

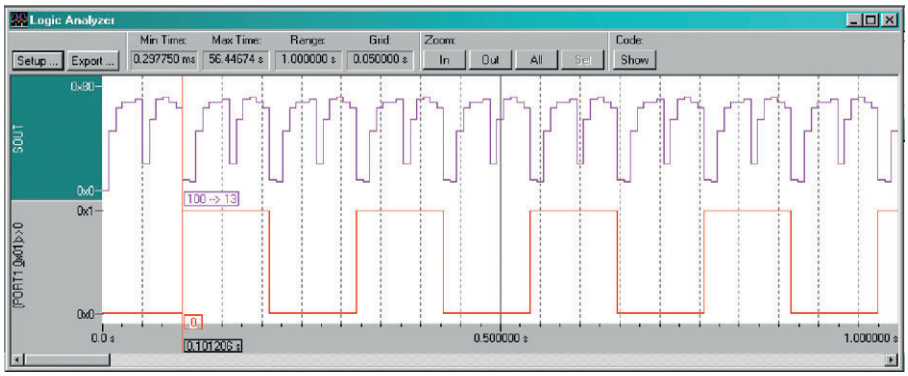

Рис. 8. График для сигналов *PORT1* и *SOUT*

символ (например, нажав клавишу Enter); программа выдаст очередное сообщение и перейдёт в режим ожидания. Остановим программу.

Графики сигналов в логическом анализаторе примут вид (при достаточно большом увеличении), показанный на рисунке 9. Обратите внимание, что заголовок одного из сигналов, отображающихся в левой части окна, выделен цветом, поскольку к его изменениям привязана линия среза времени. Выделение сигнала производится щелчком мыши по его заголовку. Линия среза перемещается по графику под управлением клавиш «стрелка влево» и «стрелка вправо», при этом перемещение линии среза происходит скачкообразно от одного изменения состояния сигнала до другого. Такое управление достаточно удобно использовать для точного позиционирования линии среза.

Рассмотрим подробно изменения сигналов на графиках. Цифрой 1 обозначен момент ввода символа пользователем. Программа переходит из режима ожидания к активному функционированию и дублирует введённый символ на линии *SOUT* (момент времени 2). Далее программа переходит к началу выполняемого цикла и изменяет состояние сигнала *PORT1* на противоположное значение (момент времени 3). Затем следует подготовка данных функцией *printf* и выдача самих данных (момент времени 4).

Можно заметить неточность в длительности отображения состояния входного сигнала *SIN*: создаётся впечатление, что скорость ввода данных через последовательный порт значительно больше скорости вывода, хотя скорости должны совпадать.

В процессе выполнения программы наберём в командной строке:

```
SIN = 'Q'
```

В результате символ '*Q*' будет обработан программой, аналогично символам, вводимым в окне *Serial #1*. Введём следующие команды:

```
INCLUDE uart.ini
DEFINE BUTTON "Uart_in",
"uart_in();"
```

Теперь вызовем панель кнопок пользователя и убедимся в наличии нашей кнопки с именем *Uart_in*, как показано на рисунке 10. Нажатие на кнопку вызовет обработчик нажатия (процедура *uart_in()*), который, в свою очередь, введёт очередной символ в последовательный порт.

Рассмотрим ещё одну интересную возможность языка сценариев, для этого добавим в код программы объявление переменной:

```
unsigned int val;
```

Вместо вызова функции getchar в конце программы вставим строку:

```
printf("Введённое значение равно
%u.\n", val);
```

Создадим в файле сценариев следующую процедуру:

```
signal void on_read (unsigned
long adr) {
```

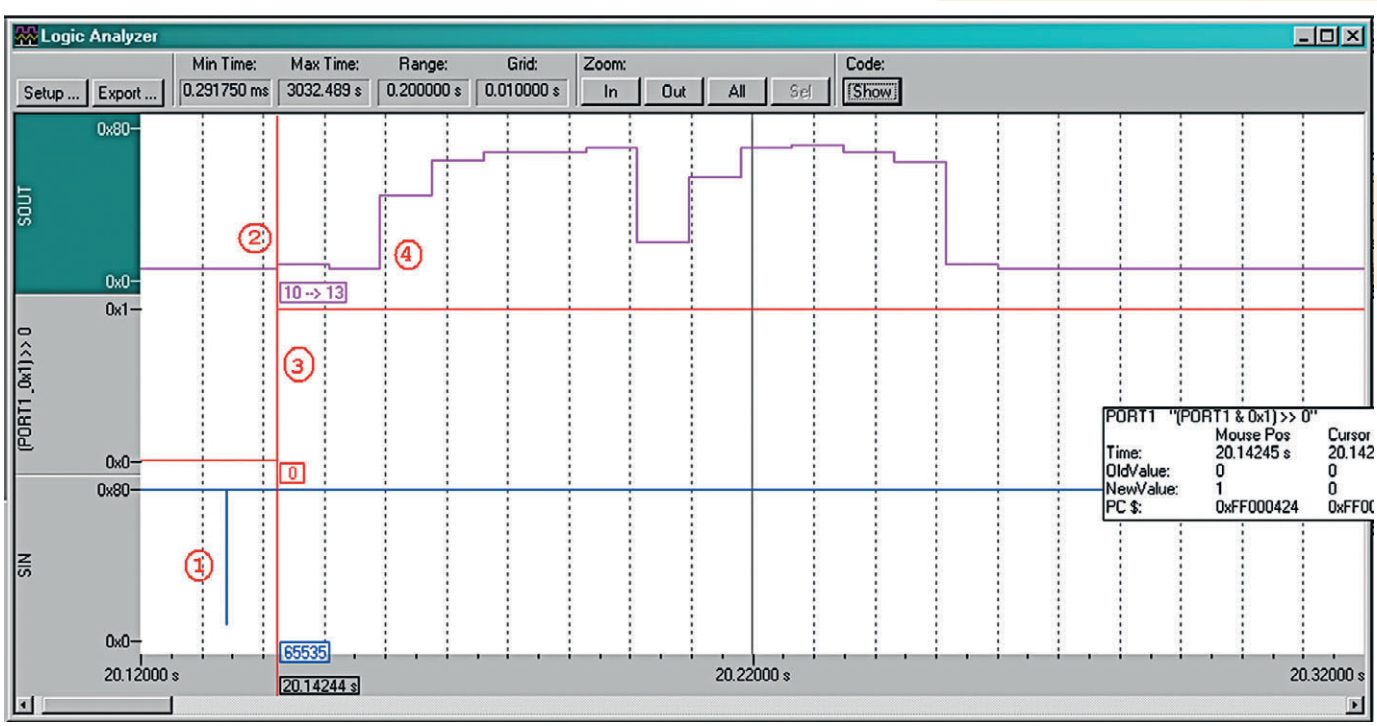

Рис. 9. Графики сигналов в логическом анализаторе

```
unsigned int val;
printf ("Отследить доступ к па-
мяти по адресу 0x%X\n", adr);
while (1) {
rwatch (adr); // ожидаем доступа
к переменной по адресу
val = getint ("Введите целое
значение");
if (0 == val)
_break_ = 1; // выходим из бес-
конечного цикла,
останавливая выполнение программы
else {
_WBYTE (adr, (val>>8)); // пере-
дать в программу
старший байт
_WBYTE (adr+1, (val&0xff)); //
теперь младший байт
}
}
```

Теперь скомпилируем программу, запустим отладчик и введём команды:

```
INCLUDE uart.ini
on_read(&val)
```

Запускаем программу. На экране появляется модальное окно для ввода целых значений. Ввод значений можно осуществлять как в десятеричной, так и в шестнадцатеричной системе счисления. После ввода значения будет выполнен основной цикл программы, и в окне *Serial #* появятся соответствующие сообщения. Для прекращения вывода модального окна достаточно ввести нулевое или пустое значение.

Рассмотрим подробно механизм работы процедуры сценария. Прежде всего, отметим использование процедуры *rwatch*, которая принимает адрес на область памяти или регистр и ожидает, пока отлаживаемая программа не обратится за значением, содержащимся по этому адресу. Механизм работы сценария напоминает работу точки останова типа A, но отличается возможностью произвести автоматическое вычисление запрашиваемого значения.

В нашем случае вычисление значения сводится к отображению стандартного диалогового окна запроса, что само по себе является полезной возможностью языка сценариев. Неудобство заключается в том, что диалоговое окно

Рис. 10. Дополнительная кнопка

выводится в модальном режиме и блокирует все возможности управления программой, поэтому в коде процедуры сценария предусмотрена возможность остановки выполнения программы путём установки встроенной переменной *_break_* в единичное значение. Установка переменной *_break_* в ненулевое значение в процедуре сценариев или их командной строке приводит к немедленной остановке выполнения отлаживаемой программы.

Таким образом, мы рассмотрели наиболее интересные и полезные для практического использования возможности отладчика среды разработки Keil. Надеемся, что данная статья окажет действенную помощь в использовании неочевидных возможностей, предоставляемых отладчиком Keil uVision.