

## Глава 1. Первый проект

Данный документ представляет собой вольный перевод книги "*Programming 16-Bit PIC Microcontrollers in C. Learning to Fly the PIC24*" (Lucio Di Jasio, 2007). В процессе перевода мы обошли стороной рассуждения автора о лётчиках, а также некоторые шуточные изречения, выделив основную суть. В итоге текст получился более официальным, и было решено оформить перевод в виде уроков. Все замечания и комментарии вы можете присылать на e-mail: [picozoid@mail.ru](mailto:picozoid@mail.ru)

<b>План урока.....</b>	<b>1</b>
<b>Перечень необходимого.....</b>	<b>1</b>
<b>Урок.....</b>	<b>2</b>
Компиляция и компоновка.....	3
Сборка первого проекта.....	4
Инициализация порта PORT.....	6
Повторная проверка порта PORTA.....	6
Тестирование порта PORTB.....	7
<b>Подводим итоги.....</b>	<b>9</b>
Замечания для экспертов по ассемблеру.....	9
Замечания для экспертов по микроконтроллерам PIC.....	10
Замечания для экспертов по Си.....	10
<b>Советы и трюки.....</b>	<b>10</b>
<b>Упражнения.....</b>	<b>10</b>
<b>Литература.....</b>	<b>10</b>

### План урока

В этом уроке мы создадим наш первый проект для микроконтроллера PIC24 с 16-разрядной архитектурой. Возможно, для некоторых из вас это будет также и первый проект в интегрированной среде разработки MPLAB IDE и в языковой оболочке MPLAB C30.

Наверное, вам знакома фраза "Hello, World!", с которой обычно ассоциируется первый шаг в программировании. Наши уроки не будут исключением. Но так как при программировании микроконтроллеров в плане визуализации текста всё гораздо сложнее, чем в случае компьютера, на первом уроке мы ограничимся лишь самым фундаментальным типом вывода - через цифровой порт ввода/вывода. В последующих же уроках мы освоим вывод на LCD-дисплей и терминал, благодаря которым мы сможем сделать гораздо более интересные вещи, чем просто вывести "Hello, World!".

### Перечень необходимого

Для дальнейшего изучения нам потребуются:

- MPLAB IDE, бесплатная интегрированная среда разработки;
- MPLAB SIM, программный симулятор;
- MPLAB C30, компилятор языка Си (бесплатная "студенческая" версия).

Теперь создадим новый проект в MPLAB IDE:

1) Выберите меню "*Project→Project Wizard*". Запустится Мастер создания нового проекта, который автоматически проведёт вас через указанные далее шаги;

2) Выберите устройство "PIC24FJ128GA010" и нажмите "*Next*";

3) Выберите компилятор "MPLAB C30 Compiler Suite" и нажмите "*Next*";

4) Создайте новую папку и назовите её "Hello", назовите проект как "Hello Embedded World" и нажмите "*Next*";

5) Затем просто нажмите "*Next*" (в данном случае в копировании каких-либо исходных файлов из предыдущих проектов нет необходимости);

6) Нажмите "*Finish*", чтобы завершить создание проекта. Затем выполните несколько следующих шагов;

7) Откройте окно редактора;

8) Напишите три строки-комментария:

```
//  
// Hello Embedded World!  
//
```

9) Выберите меню "*File*→*Save As*" и сохраните созданный файл с именем "hello.c".

10) Выберите меню "*Project*→*Save*", чтобы сохранить проект.

## Урок

Приступим к написанию нашего кода. Первая строка будет следующей:

```
#include <p24fj128ga010.h>
```

Это не совсем синтаксис Си, это директива препроцессора, сообщающая компилятору, что в это место программы нужно поместить содержимое файла p24fj128ga010.h. В этом файле содержится длинный список имён (и размеров) всех внутренних регистров специального назначения (*Special Function Registers* - *SFRs*) выбранной модели микроконтроллера (в данном случае это PIC24FJ128GA010). Если имя файла выбрано правильно, то в нём будут перечислены все имена регистров, имеющиеся в техническом описании на устройство. Этот файл является текстовым, его можно открыть в окне редактора MPLAB. Вот пример его содержимого, в котором определены счётчик команд и некоторые другие регистры специального назначения:

```
...  
extern volatile unsigned int PCL __attribute__((__sfr__));  
extern volatile unsigned char PCH __attribute__((__sfr__));  
extern volatile unsigned char TBLPAG __attribute__((__sfr__));  
extern volatile unsigned char PSVPAG __attribute__((__sfr__));  
extern volatile unsigned int RCOUNT __attribute__((__sfr__));  
extern volatile unsigned int SR __attribute__((__sfr__));  
...
```

Теперь вернёмся к нашему "hello.c" и добавим в него ещё несколько строк, представляющих собой код функции main():

```
main()  
{  
  
}
```

Теперь мы получили завершённый, но пустой и совершенно бесполезный код, - программу на языке Си. Между двумя фигурными скобками мы вскоре разместим несколько первых команд нашего встраиваемого приложения, а пока немного теории.

Независимо от того, где эта функция расположена в файле, функция main() - это то место программы, куда устанавливается программный счётчик микроконтроллера сразу же после включения питания, а также после каждого последующего сброса.

Небольшое пояснение: перед входом в функцию main() микроконтроллер сначала выполнит короткий инициализирующий код, автоматически добавляемый компоновщиком. Этот код известен как код C0. Он выполняет такие рутинные операции, как, например, инициализация стека микроконтроллера.

Итак, продолжим. Мы упоминали, что в нашем первом уроке мы попробуем подать сигнал на одну или несколько линий ввода/вывода, а именно выберем линии RA0-7 порта PORTA. На ассемблере для передачи на порт константного значения мы бы использовали пару команд mov. На Си это сделать проще - достаточно воспользоваться оператором присваивания:

```
#include <p24fj128ga010.h>
main()
{
    PORTA = 0xff;
}
```

Во-первых, обратите внимание на то, что каждая отдельная инструкция (*statement*) в Си заканчивается символом ";". Во-вторых, заметьте, как наше выражение похоже на математическое равенство... но оно им не является!

Оператор присваивания состоит из двух частей: правой и левой. Сначала вычисляется правая часть, затем полученное значение (в нашем случае просто константа) передаётся в левую часть, которая выступает в качестве принимающего контейнера (в нашем случае это 16-разрядный регистр специального назначения микроконтроллера - имя, предопределённое в файле `.h`).

**Примечание:** В языке Си существует возможность указания чисел в синтаксисе разных систем счисления. По умолчанию числа указываются в десятичной системе. Для задания числа в шестнадцатеричной системе используется префикс "0x", для чисел в двоичной системе - "0b", для восьмеричной - просто одиночный "0".

**Замечание от Greg:** К сожалению, в языке Си префикс "0b" не является стандартным для обозначения двоичной системы. В целях дальнейшей совместимости написанных программ с другими компиляторами такой формы записи желательно избегать. Например, вместо `PORTA = 0b11001000;` использовать `PORTA = 1<<7 | 1<<6 | 1<<3;`.

## Компиляция и компоновка

Теперь, когда единственная в нашей первой программе функция `main()` написана, возникает вопрос - как нам перевести текст программы на Си в двоичный исполняемый код?

С помощью MPLAB IDE это можно сделать всего одним нажатием мыши! Эта операция называется "*Project Build*". При этом выполняется довольно длинная и сложная последовательность действий, которые можно сгруппировать в два основных этапа:

1) **Компиляция:** вызывается компилятор (*compiler*) Си и генерирует файл объектного кода (`.o`), который пока ещё не является исполняемым. Хотя основная часть генерации объектного кода выполнена, все адреса функций и переменных по-прежнему остаются неопределёнными (полученный код также называют "перемещаемым объектным кодом"). Если файлов с исходным кодом несколько, эта операция повторяется для каждого из них;

2) **Компоновка:** вызывается компоновщик (*linker*), и для каждой функции и переменной определяется надлежащее место в пространстве памяти. В данный момент также можно добавить любое количество файлов объектного кода прекомпилятора и стандартных библиотечных функций. Компоновщик генерирует несколько выходных файлов, среди них находится исполняемый файл (`.hex`).

Всё это происходит очень быстро, едва вы выберете "*Build All*" в меню "*Project*".

Существуют и альтернативные методы вызова компилятора и компоновщика с помощью командной строки без использования MPLAB IDE, хотя в этом случае вам придётся обратиться к руководству пользователя по компилятору MPLAB C. Мы же, ради простоты, будем придерживаться работы с интерфейсом MPLAB IDE.

Чтобы MPLAB знал, какие файлы нужно компилировать, мы должны добавить их имена в список файлов с исходным кодом проекта "*Source Files List*" (в нашем случае это файл "`hello.c`").

Чтобы компоновщик смог назначить правильные адреса для каждой переменной и функции, мы должны сообщить MPLAB имя устройство-зависимого файла сценария компоновщика (*linker script*) (`.gld`). Подобно тому, как подключаемый файл (`.h`) сообщает компилятору об именах (и размерах) SFR-регистров для заданного устройства, файл `.gld` сообщает компоновщику об их предопределённом

расположении в памяти, а также такую важную информацию, как полный объём флэш-памяти, ОЗУ (*RAM*) и их диапазоны адресов.

Файл сценария компоновщика - это обычный текстовый файл, который можно открыть и посмотреть в окне редактора MPLAB.

Ниже приведён фрагмент файла `p24fj128ga010.gld`, в котором определены адреса программного счётчика и нескольких других SFR:

```

...
_PCL      = 0x2E;
_PCL      = 0x2E;
_PCH      = 0x30;
_PCH      = 0x30;
_TBLPAG   = 0x32;
_TBLPAG   = 0x32;
_PSVPAAG  = 0x34;
_PSVPAAG  = 0x34;
_RCOUNT  = 0x36;
_RCOUNT  = 0x36;
_SR       = 0x42;
_SR       = 0x42;
...

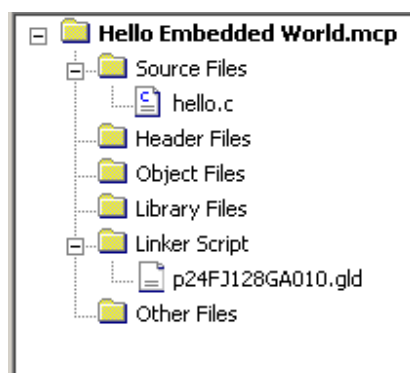
```

## Сборка первого проекта

Для завершения нашего первого демонстрационного проекта понадобится выполнить ещё несколько шагов:

- 1) Добавьте текущий файл с исходным кодом в список "**Project Source Files**":
  - a) Если окно проекта ещё не открыто, откройте его командой меню "*View*→*Project*";
  - b) Щёлкните правой кнопкой мыши в окне редактора;
  - c) В появившемся меню выберите "*Add to project*".
- 2) Добавьте к проекту файл сценария компоновщика:
  - a) Щёлкните правой кнопкой мыши в окне проекта и выберите "*Add file*";
  - b) Найдите в подкаталоге MPLAB `support/gld` файл "`p24fj128ga010.gld`" и выберите его.

Теперь окно вашего проекта будет выглядеть как на **Рис. 1-1**.



**Рис. 1-1.** Окно проекта в MPLAB IDE, настроенное на проект "Hello Embedded World"

3) Выберите команду "*Project*→*Build*" и наблюдайте, как работают компилятор и компоновщик, генерируя исполняемый код и сообщения в окне "*Output*→*Build*" (см. **Рис. 1-2**).

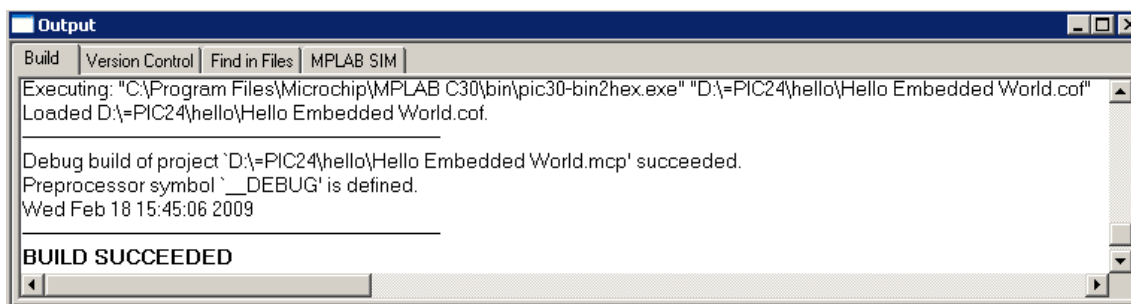


Рис. 1-2. Окно "Output" и вкладка "Build" после успешной сборки проекта

4) Выберите "Debugger→Select Tool→MPLAB SIM", чтобы активировать симулятор, – наше главное отладочное средство для этого урока.

Давайте перед выполнением кода откроем ещё и окно наблюдения "Watch" и добавим в него SFR-регистр PORTA (напечатайте или выберите PORTA в выпадающем списке с SFR, а затем нажмите кнопку "Add SFR") (см. Рис. 1-3).

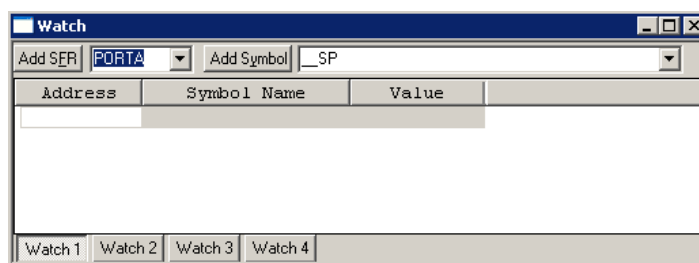


Рис. 1-3. Окно "Watch"

5) Нажмите кнопку "Reset" (🔄) симулятора (или меню "Debugger→Reset") и понаблюдайте за значениями битов порта PORTA. После сброса они должны быть установлены в ноль. Затем поместите курсор на строку, содержащую оператор присваивания значения порту (в функции main), щёлкните правой кнопкой мыши и выберите "Run to Cursor".

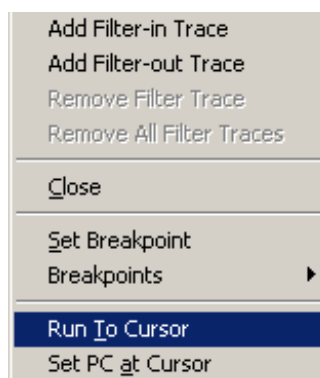


Рис. 1-4. Контекстное меню редактора MPLAB IDE (вызывается по правой кнопке мыши)

Это позволит вам пропустить весь код инициализации и попасть непосредственно на начало вашего кода.

6) Теперь пройдите один шаг с помощью функции "Step-Over" (⏩) или "Step-In" (⏪) для выполнения одного (и единственного) предложения нашей программы и понаблюдайте за изменением содержимого PORTA в окне "Watch". А там ничего и не изменилось!

## Инициализация порта PORT

Самое время открыть спецификацию (*datasheet*) на микроконтроллер PIC24FJ128GA (Глава 9) и детально ознакомиться с портами ввода/вывода.

К примеру, если мы посмотрим на PORTA то увидим что это довольно функционально нагруженный 16-разрядный порт.

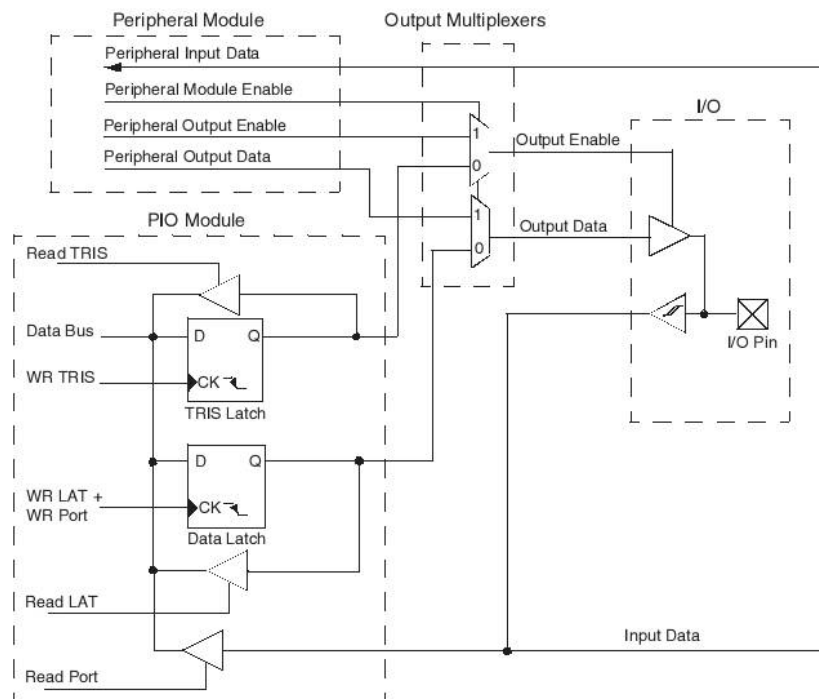


Рис. 1-5. Схема типичного порта ввода/вывода PIC24

По схеме расположения выводов из спецификации видно, что на каждый вывод порта мультиплексировано множество периферийных модулей. Также можно определить направление всех выводов, устанавливаемое по умолчанию после условия сброса: они настроены как входы (стандарт для PIC-микроконтроллеров). SFR-регистр `TRISA` управляет направлением каждого вывода порта PORTA. Поэтому, если мы хотим, чтобы состояние порта PORTA изменилось, необходимо поменять направление всех его выводов, а для этого нам нужно добавить в нашу программу ещё один оператор присваивания:

```
#include <p24fj128ga010.h>
main()
{
    TRISA = 0;          // настроить все выводы PORTA как выходы
    PORTA = 0xff;
}
```

## Повторная проверка порта PORTA

- 1) Снова выполните сборку проекта.
- 2) Установите курсор на строку с `TRISA`.
- 3) Выполните команду "*Run to Cursor*".
- 4) Выполните пару одиночных шагов и ... у нас получилось!

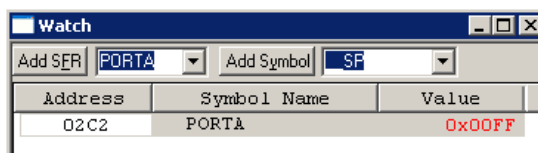


Рис. 1-6. Часть окна "Watch": содержимое PORTA изменилось!

Вы увидите, что содержимое PORTA изменилось на 0x00FF и в окне "Watch" выделилось красным цветом!

Наш выбор порта PORTA был продиктован частично тем, что он первый по алфавиту, и частично тем, что на популярной демонстрационной плате Explorer16 выводы RA0-RA7 порта PORTA подключены к восьми светодиодам. Поэтому, если вы попытаете выполнить этот пример на настоящей демонстрационной плате, вы увидите, как все светодиоды будут светиться.

### Тестирование порта PORTB

В завершение нашего урока исследуем ещё один порт ввода/вывода - PORTB.

Мы просто заменим в нашей программе PORTA и TRISA на PORTB и TRISB соответственно. Теперь выполняем повторную сборку проекта и следуем по шагам, выполненным в предыдущем упражнении и видим, что... этот код не работает для PORTB!

Не волнуйтесь, всё это было сделано специально, чтобы показать вам некоторые проблемы, связанные с переходом на PIC24.

Если мы заглянем в спецификацию и более детально изучим схему расположения выводов PIC24, то увидим два фундаментальных различия между архитектурой 8-разрядных микроконтроллеров и новой архитектурой PIC24:

1) Большинство выводов PORTB мультиплексированы с аналоговыми входами модуля АЦП. В 8-разрядной архитектуре для этих целей отведены выводы PORTA;

2) В PIC24, если сигнальные линии периферийного модуля мультиплексированы с линиями ввода/вывода, то при включении модуля он берёт на себя полное управление этими выводами, независимо от содержимого регистра TRISx. В 8-разрядной архитектуре в любом случае назначение правильного направления возлагалось на пользователя.

По умолчанию выводы, мультиплексированные с "аналоговыми" входами, отключены от их "цифровых" входных портов, в чём мы и убедились на нашем последнем примере. Всем выводам PORTB в PIC24FJ128GA010 по умолчанию при включении питания назначена функция аналоговых входов, поэтому чтение порта PORTB возвращает все нули. Заметьте, что при этом защёлка PORTB установилась правильно, хотя мы не можем видеть её через регистр PORTB. Чтобы её увидеть, нужно посмотреть содержимое регистра LATB.

Чтобы подключить входы PORTB к цифровым входам, нам нужно произвести некоторое действие над входами модуля АЦП. Из спецификации мы узнаём, что назначением выводов аналоговой/цифровой функции управляет регистр AD1PCFG.

Upper Byte:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG15	PCFG14	PCFG13	PCFG12	PCFG11	PCFG10	PCFG9	PCFG8
bit 15						bit 8	

Lower Byte:							
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG7	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0
bit 7						bit 0	

bit 15-0 PCFG15:PCFG0: Analog Input Pin Configuration Control bits  
 1 = Pin for corresponding analog channel is configured in Digital mode; I/O port read enabled  
 0 = Pin configured in Analog mode; I/O port read disabled, A/D samples pin voltage

Рис. 1-7. AD1PCFG - регистр настройки порта АЦП

Если присвоить "1" каждому биту этого регистра, наша программа будет завершена:

```
#include <p24fj128ga010.h>
main()
{
    TRISB = 0;           // настроить все линии порта PORTB как выходы
    AD1PCFG = 0xffff;   // настроить все линии порта PORTB как цифровые
    PORTB = 0xff;
}
```

Если мы сейчас скомпилируем её и пройдём в пошаговом режиме, то получим правильный результат.

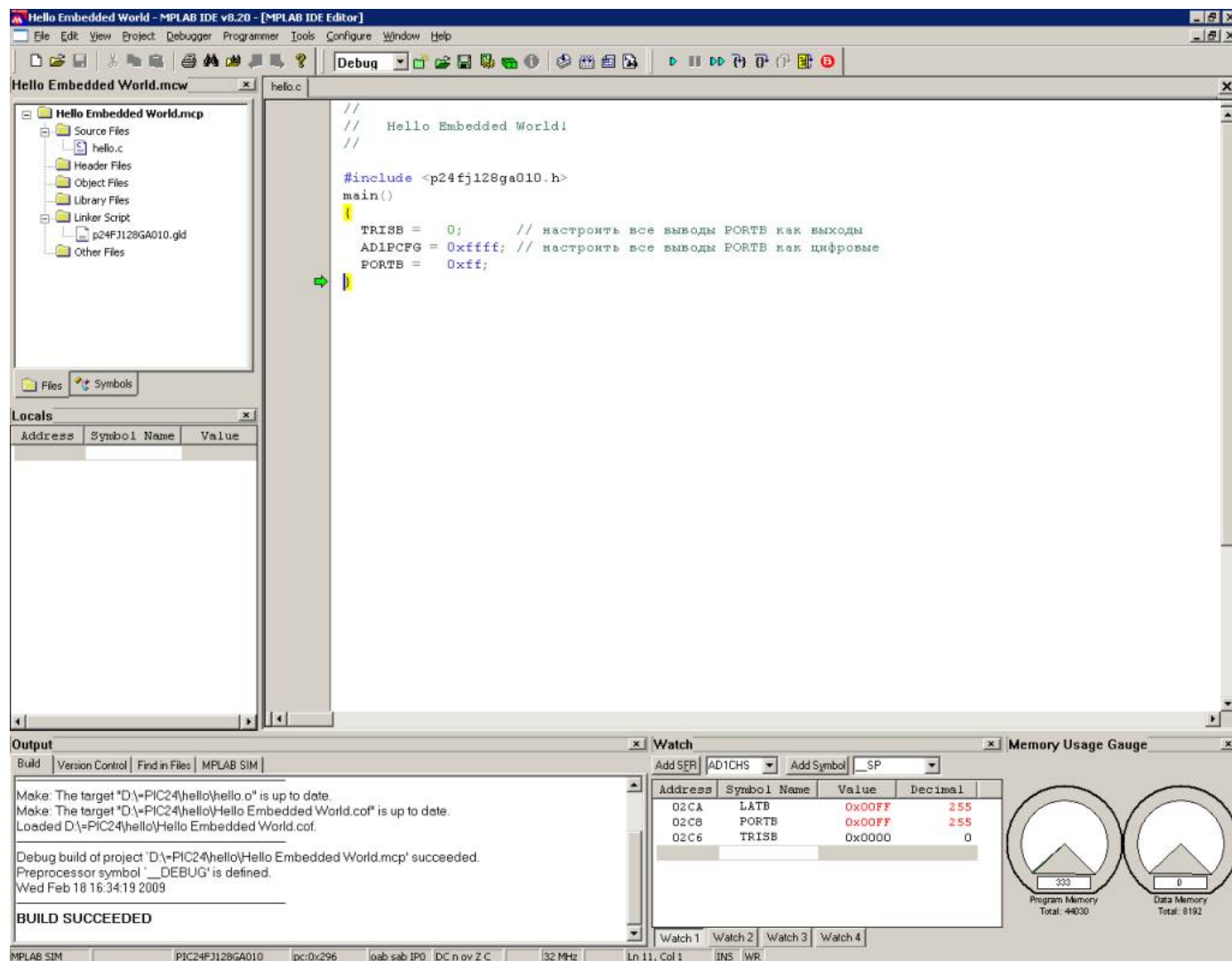


Рис. 1-8. Проект "Hello Embedded World"

**Примечание:** Чтобы настроить расположение окон Output, Watch, Memory Usage Gauge и др. как на скриншоте выше, нужно перевести их в режим *Docable* (щелчок левой кнопкой мыши по иконке в левом верхнем углу окна в строке заголовка) и перетащить к границам главного окна MPLAB IDE.

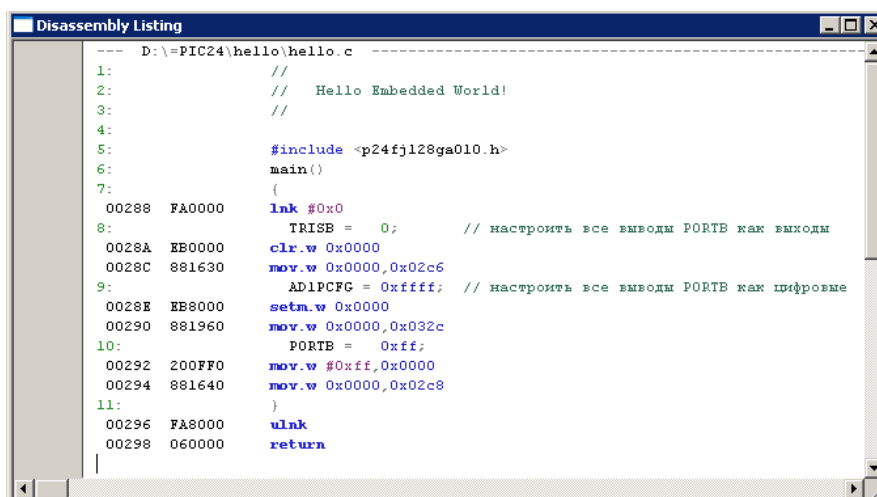
## Подводим итоги

Написание программ на языке Си для микроконтроллеров PIC может быть очень простым, или, по крайней мере, не сложнее, чем эквивалент на ассемблере. Две или три команды, в зависимости от выбранного порта, дают нам возможность управлять самым фундаментальным средством связи микроконтроллера с внешним миром - линиями ввода/вывода.

При этом имейте в виду, что компилятор С30 не умеет читать ваши мысли, и вы сами, как и на ассемблере, ответственны за правильное назначение направлений линий ввода/вывода. Именно поэтому и нужно внимательно изучать спецификацию, чтобы знать о различиях между 8-разрядными микроконтроллерами, с которыми вы, возможно, знакомы, и новым 16-разрядным семейством. Каким бы высокоуровневым не был язык Си, написание кода для встраиваемых приложений требует от нас самого близкого знакомства с мельчайшими деталями аппаратного уровня.

## Замечания для экспертов по ассемблеру

Если вам трудно слепо верить в правильность кода, генерируемого компилятором, вы в любое время можете переключиться на окно "*Disassembly Listing*" и быстро изучить созданный им код, поскольку каждая строка на языке Си указана в комментариях, предшествующих коду, который она генерирует:



```

--- D:\PIC24\hello\hello.c -----
1:      //
2:      //   Hello Embedded World!
3:      //
4:
5:      #include <p24fj128ga010.h>
6:      main()
7:      {
00288 FA0000      lnk #0x0
8:          TRISB = 0;      // настроить все выходы PORTB как выходы
0028A EB0000      clr.w 0x0000
0028C 881630      mov.w 0x0000,0x02c6
9:          AD1PCFG = 0xffff; // настроить все выходы PORTB как цифровые
0028E EB8000      setm.w 0x0000
00290 881960      mov.w 0x0000,0x032c
10:         PORTB = 0xff;
00292 200FF0      mov.w #0xff,0x0000
00294 881640      mov.w 0x0000,0x02c8
11:         }
00296 FA8000      ulnk
00298 060000      return
  
```

Рис. 1-9. Окно "*Disassembly Listing*"

Вы даже можете в пошаговом режиме пройти через весь код и произвести всю отладку из этого окна, хотя рекомендуем вам так не поступать (или ограничиться несколькими исследовательскими сессиями, которые мы проведем в первых уроках). Удовлетворите ваше любопытство, но постепенно научитесь доверять компилятору. В конечном счете, использование языка Си позволит вам увеличить вашу производительность, а также читаемость и поддерживаемость вашего кода.

В качестве последнего упражнения предлагаем вам открыть окно "*Memory Usage Gauge*" ("Индикатор использования памяти"), выбрав "*View→Memory Usage Gauge*".

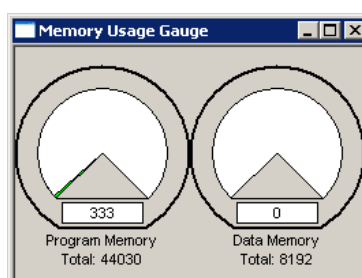


Рис. 1-10. Окно "*Memory Usage Gauge*"

Несмотря на то, что мы написали только три строки кода, объём используемой памяти программ вырос уже до 300 с лишним байтов. Однако это не показатель неэффективности языка Си, это просто минимальный блок кода, который генерируется компилятором C30 для нашего же удобства. Это и есть тот самый инициализационный код `с0`, о котором упоминалось в начале урока и мы подробнее его рассмотрим в следующих главах, когда будем изучать инициализацию переменных, распределение памяти и прерывания.

### **Замечания для экспертов по микроконтроллерам PIC**

Тем из вас, кто знаком с архитектурой семейств PIC16 и PIC18, будет интересно узнать, что большинство регистров PIC24, включая порты ввода/вывода, являются 16-разрядными.

А если вы заглянете в спецификацию на PIC24, то увидите, что большинство периферийных модулей имеют имена очень схожие, если не идентичные, модулям в 8-разрядных PIC.

### **Замечания для экспертов по Си**

Наверняка вы использовали функцию `printf` из стандартных библиотек Си. В принципе, подобные библиотеки поставляются и с компилятором C30. Но имейте в виду, что мы пишем приложения для микроконтроллеров, а не для многогигабайтных рабочих станций. Привыкайте эффективно управлять низкоуровневой периферией внутри микроконтроллеров PIC, ведь один лишь единственный вызов библиотечной функции, подобной `printf`, может добавить к вашему исполняемому коду несколько килобайтов! Не рассчитывайте, что последовательный порт и терминал, либо текстовый дисплей будут всегда вам доступны. Вместо этого воспитывайте в себе чувствительность к "весу" каждой функции или библиотеки, которую вы используете в свете ограниченных ресурсов в мире встраиваемых приложений.

### **Советы и трюки**

Семейство микроконтроллеров PIC24 основано на технологии 3В КМОП с рабочим диапазоном напряжений от 2В до 3.6В. Это требует применения соответствующего напряжения питания 3В ( $V_{dd}$ ) и ограничивает выходной ток для каждой линии I/O при выдаче лог. "1". Тем не менее, взаимодействие с традиционными 5В устройствами осуществляется достаточно просто:

1) Для обеспечения толерантности к 5В линий, настроенных как выход, используйте управляющие регистры `ODCx` (`ODCA` для `PORTA`, `ODCB` для `PORTB` и т.д.): переведите нужные линии в режим выхода с открытым стоком и подключите к ним внешние подтягивающие резисторы к питанию 5В;

2) Линии цифровых входов всегда (технологически) толерантны к 5В. Их можно непосредственно подключать к входным сигналам 5В.

Будьте осторожны с линиями I/O, которые мультиплексированы с аналоговыми входами, - они не смогут выдержать напряжения выше  $V_{dd}$ !

### **Упражнения**

Если у вас есть плата Explorer 16:

- 1) Подготовьте проект к отладке при помощи ICD2(ICD3);
- 2) Чтобы проверить пример с `PORTA`, подключите плату Explorer 16 и проверьте визуально вывод на светодиоды LED0-7;
- 3) Чтобы проверить пример с `PORTB`, подключите вольтметр (или цифровой мультиметр) к выводу `RB0` и следите за движением стрелки при пошаговом выполнении кода.

### **Литература**

Керниган В., Ричи Д., "Язык программирования Си".