Компилятор ANSI C
и среда разработки
для TI MSP430

Версия 7.XX

IMAGECRAFT C COMPILER FOR TI MSP430 v.7.XX

Перевод: Андрей Шлеенков
http://andromega.narod.ru
mailto:andromega@narod.ru

СОДЕРЖАНИЕ

1. ПРЕДИСЛОВИЕ 7

1.1. Версия, торговые марки и авторские права 7
1.1.1. Версия 7
1.1.2. Торговые марки и авторские права 7
1.2. Регистрация продукта 8
1.2.1. Использование продукта на нескольких компьютерах 8
1.3. Использование аппаратного ключа 9
1.3.1. Драйверы 9
1.3.2. Использование ключа 9
1.4. Сетевой аппаратный ключ 10
1.4.1. Отладка установки сетевого ключа 11
1.5. Соглашение о лицензировании продукта 12
1.6. О среде разработки ImageCraft 14
1.7. Поддержка 15
1.8. Обновления продукта 17
1.9. Типы и расширения файлов 18
1.9.1. Входные файлы 18
1.9.2. Выходные файлы 18
1.10. Прагмы и расширения 19
1.10.1. #pragma 19
1.10.2. Комментарии C++ 20
1.10.3. Двоичные константы 20
1.10.4. Встроенный ассемблерный код 20
1.11. Конвертирование из других ANSI C компиляторов 21
1.12. Оптимизация 22
1.12.1. Компрессор кода – Code Compressor™ 23
1.12.2. Машинно-независимый оптимизатор 23
1.13. Благодарности 24

2. ВВЕДЕНИЕ 25

2.1. Быстрый старт 25
2.1.1. Старт нового проекта 25
2.2. Анатомия Си программы 26
2.3. Краткий обзор среды разработки 27
2.4. Использование менеджера проекта 28

3. СРЕДА РАЗРАБОТКИ 29

3.1. Концепция среды разработки 29
3.2. Управление проектом 30
3.2.1. Создание нового проекта 30
3.2.2. Опции проекта 30
3.2.3. Компиляция проекта 30
3.2.4. Перемещение проекта 31
3.3. Список файлов проекта и окно обозревателя кода 32
3.3.1. Обозреватель кода 32
3.4. Компиляция отдельного файла 33
3.5. Редактор 34
3.5.1. Внешние редакторы 34
3.6. Application Builder 35
3.7. Окно состояния 36
3.8. Эмулятор терминала 37

4. СИСТЕМА МЕНЮ ... 39
4.1. Всплывающие меню ... 39
4.2. Меню File ... 40
4.3. Меню Edit ... 41
4.4. Меню Search ... 42
4.5. Меню View ... 43
4.6. Меню Project ... 44
4.7. Меню RCS ... 45
4.7.1. RCS функции среды разработки ... 45
4.8. Меню Tools ... 46
4.9. Меню Terminal ... 47
4.10. Опции компилятора ... 48
4.11. Опции компилятора: Пути ... 49
4.12. Опции компилятора: Компилятор ... 50
4.13. Опции компилятора: Целевое устройство ... 51
4.14. Опции среды разработки ... 52
4.14.1. Опции просмотра обозревателя кода ... 52
4.14.2. Предпочтительный редактор ... 52
4.15. Опции терминала ... 54
4.16. Опции редактора и печати ... 55
4.16.1. Опции ... 55
4.16.2. Контекстная подсветка ... 56
4.16.3. Назначения клавиш ... 56
4.16.4. Шаблоны кода ... 56
5. ПРЕПРОЦЕССОР Си ... 57
5.1. Диалекты препроцессора Си ... 57
5.2. Предопределенные макросы ... 58
5.3. Поддерживаемые директивы ... 59
5.3.1. Макроопределения ... 59
5.3.2. Условная обработка ... 59
5.3.3. Дополнительно ... 60
5.4. Строковые литералы и склейка лексем ... 61
6. КРАТКОЕ ОПИСАНИЕ Си ... 63
6.1. Введение ... 63
6.1.1. Стандарты Си ... 63
6.1.2. Порядок трансляции и препроцессор Си ... 63
6.1.3. Структура исходного текста и заголовочные файлы ... 64
6.1.4. Глобальные и локальные переменные, параметры ... 65
6.2. Объявление ... 66
6.2.1. Чтение объявлений ... 66
6.2.2. Атомарность доступа ... 67
6.2.3. Указатели и массивы ... 67
6.2.4. Типы структура и объединение ... 67
6.2.5. Прототип функции ... 68
6.3. Выражения и повышение типа ... 69
6.3.1. Завершение точкой с запятой ... 69
6.3.2. Левое и правое значения ... 69
6.3.3. Целые константы ... 69
6.3.4. Выражения ... 70
6.3.5. Операции ... 71
6.4. Операторы ... 73
6.4.1. Оператор выражение ... 73
6.4.2. Составной оператор ... 73
6.4.3. Оператор If ... 73
6.4.4. Оператор While ... 73
6.4.5. Оператор For ... 73
6.4.6. Оператор Do ... 73
6.4.7. Оператор Break ... 74
6.4.8. Оператор Continue ... 74

6.4.9. Оператор Goto 74
6.4.10. Оператор Return 74
6.4.11. Оператор Switch 74

7. БИБЛИОТЕКА Си И ФАЙЛ ЗАПУСКА 75

7.1. Замена библиотечной функции 75
7.2. Файл запуска 76
7.3. Общее описание библиотеки Си 77
7.3.1. Исходный код библиотеки 77
7.3.2. Функции, специфичные для MSP430 77
7.3.3. Другие заголовочные файлы....77
7.4. Функции символьного типа 78
7.5. Математические функции с плавающей точкой 79
7.6. Стандартные функции ввода/вывода.... 81
7.6.1. Вывод возврата каретки 81
7.6.2. Использование Printf с несколькими устройствами 81
7.6.3. Список стандартных функций ввода/вывода 81
7.7. Стандартная библиотека и функции памяти 84
7.8. Строковые функции 86
7.9. Функции с переменными параметрами 88

8. ПРОГРАММИРОВАНИЕ MSP430 89

8.1. Доступ к специфическим ресурсам MSP430.... 89
8.2. Манипуляция битами 90
8.2.1. Битовые макросы 90
8.2.2. Манипуляция битами, bit-переменная и битовое поле....90
8.3. Встроенный ассемблер 91
8.4. Регистры ввода/вывода 92
8.5. Абсолютная адресация памяти 93
8.5.1. Использование ассемблерного модуля....93
8.5.2. Использование встроенного ассемблера....93
8.6. Си-задачи 94
8.6.1. Мониторы....94
8.7. Обработка прерываний.... 95
8.7.1. Си обработчики прерываний 95
8.7.2. Модифицирование сохраненного регистра SR....95
8.7.3. Ассемблерные обработчики прерываний....95

9. АРХИТЕКТУРА ВРЕМЕНИ ИСПОЛНЕНИЯ 97

9.1. Размеры типов данных 97
9.2. Интерфейс ассемблера и соглашения о вызовах.... 98
9.2.1. Внешние имена 98
9.2.2. Регистры аргументов и возвращаемых значений....98
9.2.3. Предохраняемые регистры 98
9.2.4. Volatile регистры....98
9.2.5. Обработчики прерываний....99
9.3. Функции, возвращающие нецелые значения 100
9.3.1. Возвращаемые значения типов Long и Float 100
9.3.2. Передача структуры по значению....100
9.3.3. Возврат структуры по значению....100
9.4. Машинные процедуры Си.... 101
9.5. Области программы 102
9.5.1. Память только для чтения....102
9.5.2. Память данных 102
9.5.3. Области, определяемые программистом....102
9.6. Функции стека и кучи.... 104

10. ОТЛАДКА 105

10.1. Общие приемы отладки.... 105
10.1.1. Тестирование логики программы....107
10.1.2. Файл листинга 107

10.2. Отладка в NoICE430 ... 108
10.2.1. Символы портов ... 108
11. КОМПИЛЯТОР КОМАНДНОЙ СТРОКИ ... 109
11.1. Процесс компиляции ... 109
11.2. Утилита Make ... 110
11.2.1. Параметры утилиты Make ... 110
11.3. Драйвер ... 111
11.4. Параметры компилятора ... 112
11.4.1. Параметры драйвера ... 112
11.4.2. Параметры препроцессора ... 112
11.4.3. Параметры компилятора ... 113
11.4.4. Параметры ассемблера ... 113
11.4.5. Параметры компоновщика ... 113
12. ИНСТРУМЕНТАРИЙ ... 115
12.1. Компрессор кода ... 115
12.1.1. Преимущества ... 115
12.1.2. Недостаток ... 115
12.1.3. Требования совместимости ... 115
12.1.4. Временная дезактивация компрессора кода ... 115
12.2. Система управления версиями ... 116
12.2.1. Репозиторий системы управления версиями ... 116
12.2.2. Добавление и изменение файлов репозитория ... 116
12.3. Синтаксис ассемблера ... 117
12.3.1. Имена ... 117
12.3.2. Видимость имен ... 117
12.3.3. Числа ... 117
12.3.4. Формат входного файла ... 117
12.3.5. Метки ... 117
12.3.6. Команды ... 118
12.3.7. Выражения ... 118
12.3.8. Операторы ... 118
12.3.9. “Точка” или программный счетчик ... 118
12.4. Директивы ассемблера ... 119
12.5. Команды ассемблера ... 123
12.5.1. Режимы адресации ... 124
12.6. Операции компоновщика ... 125
12.6.1. Распределение памяти ... 125
12.7. Отладочный формат ImageCraft ... 126
12.7.1. Отладчик NoICE ... 126
12.8. Библиотекарь ... 127
12.8.1. Компиляция файла в библиотечный модуль ... 127
12.8.2. Распечатка содержания библиотеки ... 127
12.8.3. Добавление или замена модуля ... 127
12.8.4. Удаление модуля ... 127

1. ПРЕДИСЛОВИЕ

1.1. Версия, торговые марки и авторские права

1.1.1. Версия

Этот печатный документ сгенерирован из документа интерактивной справки, включенного в программный продукт версии 7.XX. Так как мы непрерывно обновляем нашу продукцию, иногда печатный документ отстает от поставляемого программного продукта. В случае сомнений, за новейшей информацией следует обращаться к интерактивной документации. Данный документ последний раз обновлялся 26 февраля 2006 г.

1.1.2. Торговые марки и авторские права

ImageCraft, ICC08, ICC11, ICC12, ICC16, ICCAVR, ICCtiny, ICCM8C, ICC430, ICCV7 for AVR, ICCV7 for ARM, ICCV7 for 430, MIO (Machine Independent Optimizer) and Code Compressor ™, ImageCraft Creations Inc. Авторские права на этот документ © 1999-2006 принадлежат компании ImageCraft Creations Inc. Все права зарезервированы.

Atmel, AVR, MegaAVR and tinyAVR ® Atmel Corporation.

Motorola, HC08, MC68HC11, MC68HC12 and MC68HC16 ® Motorola Inc. and Freescale Semiconductor Inc.

MSP430 ® Texas Instruments Inc.

ARM, Thumb, Cortex ® ARM Inc.

Авторские права © 1999-2006 ImageCraft Creations Inc. Все права зарезервированы.

Все торговые марки принадлежат их соответствующим владельцам.

1.2. Регистрация продукта

Вместо описанной ниже схемы лицензирования программного обеспечения может использоваться аппаратный ключ. См. 1.3. Использование аппаратного ключа.

ПОЖАЛУЙСТА, ПРОЧТИТЕ ЭТО ПЕРЕД УСТАНОВКОЙ!

Программный продукт использует разные ключи лицензирования для разрешения разных возможностей. По умолчанию, создаваемый программный код ограничен объемом 4 Кбайт. Если вы устанавливаете программное обеспечение впервые, программный продукт будет полностью функционален (как при стандартной лицензии) в течение 45 дней, после чего объем создаваемого программного кода будет ограничен постоянно. Версия с ограничением кода может быть использована только в личных некоммерческих целях. После приобретения лицензии, ее необходимо зарегистрировать при помощи команды меню *Help>Register Software*. Пожалуйста, следуйте инструкциям в диалоговом окне.

Если вы имеете действующую лицензию, то вы можете производить обновления до последней версии программного продукта, загружая последнюю демо-версию и устанавливая ее в тот же самый каталог, где установлена ваша текущая версия.

В случае каких-либо неполадок, при необходимости переустановки программного продукта и при потере ключа лицензирования, свяжитесь с нами, и мы дадим вам новую копию. Мы полагаем, что возможность легко получать обновления с нашего веб-сайта перевешивает некоторые неудобства, причиняемые процессом регистрации.

1.2.1. Использование продукта на нескольких компьютерах

Если вам необходимо использовать продукт на нескольких компьютерах, таких, как настольный PC и Notebook, и если вы – единственный пользователь продукта, вы можете получить от нас отдельную лицензию. Свяжитесь с нами для получения подробностей. В качестве альтернативы, вы можете приобрести аппаратный ключ.

1.3. Использование аппаратного ключа

ICCV7 for 430 позволяет вам использовать аппаратный ключ вместо заданной по умолчанию схемы лицензирования программного обеспечения. Ключ рассчитан на параллельный порт или порт USB. Версия для параллельного порта совместима со всеми 32-х разрядными платформами Windows, но требует специальный драйвер для Windows NT/2000 и XP. Версия USB совместима со всеми 32-х разрядными Windows за исключением старых версий Windows 95 и NT 3.5x, в которых порт USB не поддерживается. Ключ USB также использует специальный драйвер на всех платформах Windows.

1.3.1. Драйверы

Чтобы установить драйверы ключа для параллельного порта, введите в командной строке следующие команды, заменив дисковод и каталог на ваш путь установки:

```
c:
cd \iccv7430\drivers;
setupdrv /par
```

Чтобы установить драйвер параллельного порта в Windows NT/2000/XP вы должны иметь права администратора.

Если вы используете ключ USB и версию Windows **отличающуюся** от XP или 2000, следуйте вышеописанным правилам, заменив последнюю команду следующей:

```
setupdrv /usb
```

В Windows XP или 2000, подключите ключ USB и подождите, пока Windows обнаружит его и запросит у вас месторасположение информационного файла драйвера. Введите команду `c:\iccv7430\drivers`, заменив `c:\iccv7430` на ваш путь установки, и Windows установит USB драйвер ключа.

Если вам необходимо деинсталлировать драйвер, перейдите в тот же самый каталог, и введите:

```
setupdrv /ufull
```

1.3.2. Использование ключа

Для использования аппаратного ключа, просто подсоедините его перед вызовом среды разработки, и схема защиты продукта позволит вам полнофункциональную работу. Ключ должен оставаться присоединенным при компиляции и генерации проекта. Если аппаратный ключ не используется, применяется схема лицензирования по умолчанию. См. 1.2. Регистрация продукта.

1.4. Сетевой аппаратный ключ

В дополнение к ключу для одной лицензии, вы можете также приобрести сетевой ключ для управления несколькими лицензиями. В этом случае, программный продукт может быть установлен на любое число рабочих станций сети, но только определенное число их может использовать продукт одновременно. Мы имеем инсталляции, позволяющие лицензировать работу от одного до 50 пользователей.

Все ваши машины должны принадлежать одной сети. Вы должны назначить одну машину, сервером ключа и подключить сетевой ключ к данной машине. Вы должны также создать на сервере ключа одну директорию, доступную на чтение/запись для клиентских машин. Сервер ключа и клиентские машины могут использовать разные комбинации 32-х битных операционных систем Windows. Для установки сетевого ключа следуйте следующим шагам:

1. Инсталлируйте ICCV7 for 430 на сервер ключа и на все машины, где вы желаете использовать данный продукт.

На сервере ключа:

1. Следуйте инструкциям, описанным выше в 1.3. Использование аппаратного ключа для инсталляции драйвера аппаратного ключа.
2. Запустите из командной строки программу `c:\iccv7430\drivers\ddnet.exe`. Она запросит у вас директорию для хранения лицензионной информации. Укажите путь, где клиентские машины имеют полные права на чтение/запись.

   Эта информация сохраняется в файле `ddnet.ini` в директории Windows (например, `c:\windows` или `c:\winnt`). Если вам необходимо изменить расположение директории, вы можете отредактировать этот файл непосредственно, или уничтожить его и запустить `ddnet` снова.

   Вы должны запускать `ddnet.exe` каждый раз при перезагрузке сервера, например, создав ярлык в программной группе Автозагрузка системы Windows. В качестве альтернативы, если вы используете Windows NT/2000/XP, вы можете инсталлировать `ddnet` как сервис, введя строку `ddnet /S`.

   Для полной деинсталляции `ddnet`, введите команду `ddnet /u`.

На клиентских машинах:

1. Переименуйте `c:\iccv7430\bin\d430.dll` в `c:\iccv7430\bin\d430.dll.orig`.
2. Переименуйте `c:\iccv7430\bin\d430net.dll` в `c:\iccv7430\bin\d430.dll`.
3. В директории Windows (например, `c:\windows` или `c:\winnt`), создайте файл по имени `icc430.ini` со следующим содержимым:

   ```
   [dinkey]
   DinkeyNetPath=\\server\drive\license_path
   ```

   `DinkeyNetPath` установлен в сетевое UNC имя пути к лицензии на сервере ключа. Вместо использования пути UNC, вы можете также использовать путь к директории диска. Например, если сервер ключа называется `foo` и путь к лицензии установлен в `c:\iccv7430\dongle_license`, вы должны написать:

   ```
   [dinkey]
   DinkeyNetPath=\\foo\c\iccv7430\dongle_license
   ```

1. Когда вы запустите ICCV7 for 430 на клиентских машинах, теперь будет использоваться сервер ключа для отслеживания количества одновременно используемых лицензий.

1.4.1. Отладка установки сетевого ключа

Из-за большого количества шагов установки сетевого ключа может произойти много неточностей и ошибок. Если вы тщательно следовали вышеприведенным инструкциям и все же встретили трудности, следуйте нижеописанным шагам для устранения проблем:

1. На машине клиента запустите программу редактора реестра `regedt32`.
2. Найдите ключ `HKEY_CURRENT_USER\Software\ImageCraft\ICCV7 for 430`.
3. Вызовите *Edit>Add Value*. Добавьте имя значения ключа `dongle` с типом данных, установленным в `REG_DWORD`. Нажмите "**OK**", и затем установите данные в 1. Затем нажмите "**OK**".
4. Когда вы запустите среду разработки, если ключ не детектируется, возникнет всплывающее окно с информацией кода ошибки. Пожалуйста, вышлите эту информацию по адресу mailto:support@imagecraft.com с описанием проблем и мы поможем вам разрешить возникшие вопросы.

1.5. Соглашение о лицензировании продукта

Приводимый далее текст является соглашением между вами, конечным пользователем, и ImageCraft. Если вы не согласны с условиями данного соглашения, пожалуйста, как можно быстрее возвратите комплект поставки, и вы получите полное возмещение стоимости.

ПРЕДОСТАВЛЕНИЕ ЛИЦЕНЗИИ. Это соглашение о лицензировании программного обеспечения ImageCraft разрешает вам использовать одну копию программного продукта ImageCraft (ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ) на любом компьютере при условии использования только одной копии одновременно.

АВТОРСКОЕ ПРАВО. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ является собственностью ImageCraft и защищено законами Соединенных Штатов Америки об авторском праве и условиями международных соглашений. Вы должны использовать ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ подобно любому другому обеспеченному авторским правом материалу (например, книге). Вы не можете копировать письменные материалы, сопровождающие ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ.

ДРУГИЕ ОГРАНИЧЕНИЯ. Вы не можете арендовать или сдавать в аренду ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ, но вы можете передавать ваши права согласно этой лицензии на постоянном основании, если вы передаете эту лицензию, ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ и все сопровождающие письменные материалы, и если вы не сохраняете никаких копий, и получатель соглашается на условия этой лицензии. Если ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ подвергалось обновлению, любая передача должна включать обновления и все предшествующие версии.

ОГРАНИЧЕННАЯ ГАРАНТИЯ. ImageCraft гарантирует, что ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ будет выполняться в основном в соответствии с сопровождающими письменными материалами и будет свободно от дефектов при нормальном использовании и обслуживании в течение тридцати (30) дней со дня получения. Любые подразумеваемые гарантии на ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ограничены 30 днями. Некоторые государства не допускают ограничений на продолжительность подразумеваемой гарантии, поэтому вышеупомянутые ограничения к вам могут не применяться. Эта ограниченная гарантия дает вам специфические действительные права. Вы можете иметь другие права, которые изменяются в зависимости от государства.

ВОЗМЕЩЕНИЕ УЩЕРБА ЗАКАЗЧИКА. Вся ответственность ImageCraft и единственное средство возмещения вам ущерба будет состоять, по выбору ImageCraft, в (a) возврате уплаченный суммы или (b) исправлении или замене ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ, которое не отвечает ограниченной гарантии ImageCraft и возвращено в ImageCraft. Эта ограниченная гарантия отменяется, если отказ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ последовал в результате несчастного случая, неосторожного обращения или неправильного использования. Любая замена ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ будет обладать гарантией на период времени, определяемый как больший из двух периодов – остаток от первоначального срока гарантии или 30 дней со дня замены продукта.

ОТСУТСТВИЕ ДРУГИХ ГАРАНТИЙ. ImageCraft отказывается от всех других гарантий, явных или подразумеваемых, включая, но, не ограничиваясь подразумеваемыми гарантиями коммерческой выгоды и пригодности для специфических целей, относительно ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ, любых сопровождающих письменных материалов и аппаратных средств.

ОТСУТСТВИЕ ОТВЕТСТВЕННОСТИ ЗА ПОСЛЕДОВАВШИЙ УЩЕРБ. Ни в каком случае ImageCraft или его дистрибьютор не будут нести ответственность за любой ущерб, какой бы он ни был (включая, но, не ограничиваясь, ущербом из-за потери прибыли, приостановки бизнеса, потери бизнес-информации или другой финансовой потери), произошедший из-за использования или неспособности использовать ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ, даже если ImageCraft уведомлялся относительно возможности такого ущерба. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ не разработано, не предназначено и не авторизовано для использования в приложениях, в которых отказ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ мог бы создавать ситуацию, способную причинить ущерб здоровью или смерть. Если вы используете ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ для любого непредназначенного для него или неавторизованного приложения, вы берете на себя обязанность возмещения убытков и должны воздерживаться от любых претензий к ImageCraft и его дистрибьюторам, даже если такие претензии утверждают, что ImageCraft был небрежен при проектировании или реализации ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

1.6. О среде разработки ImageCraft

Среда разработки ImageCraft C – это программа для разработки микроконтроллерных приложений, использующая стандарт языка ANSI C. Ее основные особенности:

- Интуитивная интегрированная среда разработки (IDE) с интегрированным редактором и менеджером проекта, предназначенная для работы в 32-х разрядных версиях Windows. Исходные файлы организуются в проекты. Редактирование и компиляция проекта могут быть выполнены полностью внутри среды. Ошибки времени компиляции отображаются в окне состояния, и простым щелчком кнопки мыши вы можете перейти в окно редактора на строки, вызвавшие ошибки. Интегрированный менеджер проекта генерирует стандартный make-файл, который вы можете просматривать и при желании использовать непосредственно.
- Среда разработки управляет ANSI C компилятором командной строки, который в работе обычно является прозрачным. Однако если вы желаете, вы можете взаимодействовать с компилятором непосредственно, используя интерфейс командный строки. Компилятор – это набор 32-х разрядных программ и распознает длинные имена файлов.

За некоторыми исключениями, этот документ не описывает язык Си в деталях и не представляет учебник Си вообще. Так как компилятор использует стандарт языка ANSI C, то вы можете использовать многочисленную литературу по языку Си, доступную в локальных книжных магазинах или в интернет-магазинах, таких, как Amazon (хотя мы рекомендуем поддержать ваши локальные независимые книжные магазины, если это возможно).

Вы также можете найти список некоторых книг, которые мы рекомендуем, на нашем веб-сайте. Однако существует гораздо больше хороших книг, так что ищите вокруг.

1.7. Поддержка

Перед тем, как связаться с нами, выясните номер версии программного обеспечения при помощи команды “About ICCV7 for 430” меню Help.

Internet и email – предпочтительные методы поддержки. Мы обычно отвечаем вам в тот же самый день и иногда даже в тот же самый час или минуту. Некоторые люди полагают, что они получат справку, только если они используют агрессивный тон или грубость. Пожалуйста, не делайте этого. Мы будем поддерживать вас самым лучшим из доступных нам способов. Мы строим нашу репутацию, на основе наилучшей поддержки. Угрозы или оскорбительный тон могут быть необходимы с другими компаниями, но мы обслуживаем наших заказчиков с уважением и все, что мы просим – отвечать в том же духе. Помощь доходит гораздо хуже, если заказчик не знает порядка обслуживания, или имеет слишком много требований, или считает, что каждая проблема – в ошибке компилятора.

Пользователь однажды прислал нам два электронных письма сразу и затем вызванный немедленно дважды отнял почти час нашего времени по телефону, потому что его простая UART программа “echo” не работала но “та же самая программа” работала под другим компилятором. Это не помогло потому, что он первоначально прислал нам неправильный исходный текст и обвинил нас в неспособности читать и выслушать его. В конечном счете, мы отметили, что он сделал простую ошибку и поместил одну из программ обработки прерывания по неправильному вектору. К сожалению, мы так и не получили даже “Спасибо”. Более грустно то, что мы можем вспомнить многих из таких заказчиков.

Электронная почта по вопросам поддержки:

support@imagecraft.com

Обновление продукта доступно бесплатно в течение шести месяцев. Файлы обновлений доступны на нашем веб-сайте:

http://www.imagecraft.com/software

Иногда мы запрашиваем, чтобы вы высылали нам ваши файлы проекта, чтобы мы могли продублировать проблему. Если возможно, пожалуйста, используйте утилиту zip для включения всех ваших файлов проекта, включая ваши собственные заголовочные файлы, в едином email вложении. Если вы не можете выслать нам по запросу весь проект, обычно достаточно, если вы сможете создать компилируемую функцию и выслать ее нам. Пожалуйста, не присылайте нам какие-либо файлы, которые не были запрошены.

Часто задаваемые вопросы (FAQ) по продукту находятся по следующему адресу:

http://www.imagecraft.com/software/FAQ.html

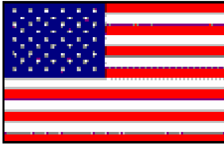
У нас имеется список рассылки, называемый icc-430, предназначенный для пользователей нашего продукта ICCV7 for 430. Для подписки посетите сайт:

http://www.dragonsgate.net/mailman/listinfo

Список рассылки не должен использоваться для общих вопросов поддержки. С другой стороны, наши активные пользователи в списках рассылки, вероятно, имеют больше специфических знаний об аппаратном обеспечении, чем мы, поскольку мы – прежде всего компания, разрабатывающая программное обеспечение. Мы можем попросить, чтобы вы присылали ваши вопросы туда.

Наш веб-сайт поддерживает страницу, где вы можете найти исходные коды, предоставленные пользователями. Вы можете посетить эту страницу, чтобы увидеть, написал ли кто-нибудь код, который вы можете использовать в ваших программах.

Если вы приобрели продукт у одного из наших международных дистрибьюторов, для поддержки вы можете сначала запросить их. Наш почтовый адрес и номера телефонов:



ImageCraft
706 Colorado Ave.
Suite 10-88
Palo Alto, CA 94303
U.S.A.
(650) 493-9326
(650) 493-9329 (FAX)

1.8. Обновления продукта

Номер версии продукта состоит из старшего и младшего номера. Пример: V7.10 состоит из старшего номера 7 и младшего номера .10. В течение первых шести месяцев со дня приобретения, вы можете обновлять продукт до последнего младшего номера версии бесплатно. Чтобы получать обновления позже, вы можете приобрести дешевый ежегодный план поддержки. Обновление до нового старшего номера версии обычно требует дополнительной платы.

Согласно схеме защиты программного обеспечения, используемой в продукте, вы получаете обновления, загружая последнюю версию "demo", доступную на нашем веб-сайте, и устанавливая ее на персональный компьютер с вашей текущей инсталляцией. Ваша существующая лицензия будет работать с новыми инсталлированными файлами. Вы можете иметь несколько версий продукта одновременно на одном компьютере. Имейте в виду, что все версии используют одни и те же записи в реестре Windows, а также любую другую системную информацию.

1.9. Типы и расширения файлов

Типы файлов определяются их расширениями. Среда разработки и компилятор основывают свои действия на типах входных файлов.

1.9.1. Входные файлы

- `.c` – специфицирует исходный файл Си.
- `.s` – специфицирует исходный файл ассемблера.
- `.h` – специфицирует заголовочный файл.
- `.prj` – файл проекта. Он создается и поддерживается средой разработки для хранения информации о проекте.
- `.src` – список файлов проекта. Он создается и поддерживается средой разработки для хранения имен файлов проекта.
- `.a` – библиотечный файл. Продукт поставляется с несколькими библиотеками. `libc430.a` – базовая библиотека, содержащая стандартную библиотеку Си и специфичные для TI MSP430 процедуры. Компоновщик связывает программу с библиотечными модулями или файлами только при наличии ссылок на них. При необходимости вы можете создавать или изменять библиотеки.

1.9.2. Выходные файлы

- `.s` – выходной ассемблерный файл. Он генерируется компилятором для каждого исходного файла Си.
- `.o` – объектный файл, получаемый трансляцией ассемблерного файла. Выходной исполнимый файл есть результат компоновки группы объектных файлов.
- `.hex` – выходной Intel HEX файл.
- `.s19` – исполнимый файл в формате Motorola/Freescale S19 Record.
- `.lst` – файл листинга с объектным кодом и конечными адресами вашей программы, собранными в один файл.
- `.mp` – файл карты, содержащий символьную информацию и размер вашей программы.
- `.dbg` – внутренний командный файл отладки ImageCraft.

Среда разработки может также создавать и другие файлы в выходном каталоге проекта.

1.10. Прагмы и расширения

1.10.1. #pragma

Компилятор распознает следующие директивы прагма:

- `#pragma interrupt_handler <func1> <func2> ...`

  Объявляет функции как обработчики прерываний, чтобы компилятор генерировал инструкции возврата из прерывания вместо инструкций обычного возврата из функции и сохранял и восстанавливал все регистры, используемые функциями. Генерирует векторы прерываний на основании номеров векторов. См. 8.7. Обработка прерываний. Эта прагма должна предшествовать определениям функций. Для определения смещений векторов используйте константы векторов, определенные в заголовочном файле `msp430x???.h`.

- `#pragma monitor <func1> <func2> ...`

  Определяет данные функции как функции "монитора" RTOS. На входе в функцию сначала сохраняется R2/SR, а затем инструкцией DINT запрещаются прерывания. Возврат из функции осуществляется при помощи инструкции `reti` (возврат из прерывания). Это позволяет функциям, нуждающимся в доступе к критическим областям ядра RTOS, сохранять состояние прерываний. См. 8.6.1. Мониторы.

- `#pragma language=extended`

  Является эквивалентом ключа расширений компилятора *Project>Options...>Compiler>Enabled Extensions*. Это обеспечивается, прежде всего, для совместимости с IAR C.

- `#pragma text:<text name>`

  Любое определение функции, появляющееся после этой прагмы размещается в области памяти `<text name>` вместо нормальной области `text`. Соответствует опции `-text:<text>` командной строки компилятора (не компоновщика). Используйте `"#pragma text:text"` для сброса области размещения в значение по умолчанию. Например:

```
#pragma text:mytext
void boot() ...          // function definition
#pragma text:text        // reset
```

  В строке ввода в меню *Project>Options...>Target>Other Options* введите:

```
-bmytext:0x????
```

  где `0x????` является начальным адресом области "`bootloader`". Заметьте, что при использовании этой прагмы вы должны сами гарантировать, что адрес, который вы определили для `text name`, не накладывается на область памяти, используемую компилятором для областей по умолчанию. См. 12.6. Операции компоновщика.

- `#pragma data:<data name>`

  Любое определение глобальной или статической переменной файла, появляющееся после этой прагмы, размещается в области памяти `<data name>` вместо нормальной области `data`. Соответствует опции командной строки `-data:<data>` компилятора (не компоновщика). Используйте `"#pragma data:data"` для сброса в значение по умолчанию. Заметьте, что при использовании этой прагмы вы должны сами гарантировать, что адрес, который вы определили для `data name`, не накладывается на область памяти, используемую компилятором для областей по умолчанию. См. 12.6 Операции компоновщика.

- `#pragma lit:<lit area>`

  Любое определение const-объекта, появляющееся после этой прагмы, располагается в `<lit area>` области памяти вместо нормальной `lit` области. Используйте "`#pragma lit:lit`" для сброса в значение по умолчанию. Заметьте, что при использовании этой прагмы вы должны сами гарантировать, что адрес, который вы определили для `lit area`, не накладывается на область памяти, используемую компилятором для областей по умолчанию. См. 12.6 Операции компоновщика.

- `#pragma abs_address:<address>`

  Не использовать перемещаемые области для функций и глобальных данных, а располагать их по абсолютным адресам, начинающимся с `<address>`. Полезно для доступа к векторам прерываний и другим аппаратным ресурсам. См. 9.5. Области программы. Параметр `<address>` является адресом байта и в настоящее время ограничен значением 64 Кбайт.

- `#pragma end_abs_address`

  Использовать для объектов нормальные перемещаемые области.

- `#pragma device_specific_function <func1> <func2> ...`

  Эта прагма используется для объявления функций, которые используют специфические для процессора регистры ввода/вывода (IO) и, следовательно, должны компилироваться, используя специфические для процессора заголовочные файлы и имена регистров ввода/вывода. Это сообщает компилятору, чтобы он декорировал имена функции суффиксом `$device_specific$` в выходном коде. Например, после следующего:

  ```
  #pragma device_specific_function putchar
  ```

  компилятор генерирует `_putchar$device_specific$` всякий раз, когда видит внешний идентификатор `putchar`. При нахождении неопределенного символа с этим суффиксом, компоновщик выводит соответствующее сообщение об ошибке.

1.10.2. Комментарии C++

Если вы разрешаете расширения компилятора (*Project>Options...>Compiler>Enable Extensions*), вы можете использовать стиль комментариев C++ в вашем исходном тексте при помощи символа //.

1.10.3. Двоичные константы

Если вы разрешаете расширения компилятора (*Project>Options...>Compiler>Enable Extensions*), вы можете использовать объявление `0b<1|0>` для определения двоичных констант. Например, `0b10101` представляет десятичное число 21.

1.10.4. Встроенный ассемблерный код

Вы можете использовать псевдофункцию `asm("string")` чтобы специфицировать встроенный код ассемблера. См. 8.3. Встроенный ассемблер.

1.11. Конвертирование из других ANSI C компиляторов

Эта страница рассматривает некоторые из вопросов, которые могут возникнуть при конвертировании исходного текста, написанного в других ANSI C компиляторах (для того же самого целевого процессора), в текст, предназначенный для компилятора ImageCraft. Если вы пишете код в стиле максимальной переносимости и приближенности к ANSI C, то, вероятно, что большая часть вашего кода будет компилироваться и работать без проблем.

- Наш тип данных `char` является беззнаковым.
- Для объявления функции как обработчик прерывания наши компиляторы используют прагму. Это почти всегда отличается от других компиляторов.
- Расширение ключевых слов. Некоторые компиляторы используют расширение ключевых слов, которые могут включать `far`, `@`, `port`, `interrupt`, и т.д. Ключевое слово `port` может быть заменено ссылкой на память. Например:

```
char porta @0x1000
```

  В общем случае, мы сторонимся расширений везде, где возможно. Наиболее частой причиной использования расширений, как нам кажется, является желание привязать заказчика к среде поставщика компилятора, а не обеспечить лучшее решение.

  Используя наш компилятор, верхний пример может быть переписан так:

```
#define PORTA (*(volatile unsigned char *)0x1000)
```

  или

```
#pragma abs_pragma:0x1000
char porta;
#pragma end_abs_pragma
```

- Соглашения о вызовах. Регистры, используемые для передачи параметров функциям, различны в разных компиляторах. Это обычно влияет только на рукописные ассемблерные функции.
- Некоторые компиляторы не поддерживают встроенный ассемблер и используют встроенные функции и другие расширения, чтобы достигнуть тех же самых целей.
- Директивы ассемблера почти всегда различны.
- Ассемблеры некоторых производителей могут использовать заголовочные файлы Си. Наш ассемблер этого не делает.
- Некоторые производители компиляторов используют структуры и битовые поля для инкапсуляции регистров ввода/вывода (IO) и могут использовать расширения для размещения их по правильным адресам памяти. Мы рекомендуем использовать свойства стандарта Си, такие как разрядные маски, и осуществлять приведение констант к адресам памяти для доступа к регистрам ввода/вывода. Наши заголовочные файлы определяют регистры ввода/вывода этим способом. См. пример:

```
#define PORTA (*(volatile unsigned char *)0x1000)
// 0x1000 is the IO port PORTA
#define bit(x) (1 << (x)) // bit operator
PORTA |= bit(0); // turn off bit 0
```

- Указатели функций имеют дополнительный уровень косвенности из-за требований компрессора кода. См. 12.1. Компрессор кода.

1.12. Оптимизация

Компиляторы ImageCraft происходят от компилятора LCC (см. 1.13. Благодарности). Как и предыдущий переносимый компилятор LCC, данный компилятор выполняет следующие оптимизации:

- Алгебраическое упрощение и свертка констант:

  Компилятор может заменять сложные алгебраические выражения более простыми выражениями (например, сложение с 0, деление на 1, и т.д.). Компилятор также вычисляет постоянные выражения и “сворачивает” их (например, “1+1” становится “2”). Обратите внимание, что в общем случае компиляторы не выполняют эти оптимизации с константами и переменными с плавающей точкой, т.к. хост операционная система (OS) и центральный процессор (CPU) (например, Windows на Intel x86) используют представление плавающей запятой с большим диапазоном и точностью чем целевой процессор (микроконтроллер). Следовательно, если бы эти оптимизации были выполнены со значениями с плавающей точкой, они могли бы дать результаты, отличающиеся от операций, которые должны быть выполнены целевым устройством.

- Удаление общего подвыражения в базовом блоке.

  Выражения, которые многократно используются внутри базового блока (то есть, прямая последовательность кода без переходов), могут кэшироваться компилятором без повторного вычисления.

- Оптимизация переключателя Switch.

  Компилятор анализирует значения переключателя и генерирует код, использующий комбинацию двоичного поиска и таблиц переходов. Таблицы перехода эффективны для плотно упакованных значений переключателя, а двоичный поиск размещает прямую быструю таблицу переходов. В случае, если значения сильно отличаются или немногочисленны, выполняется простой поиск "if-then-else".

Выходной генератор кода компилятора использует методику называемую перезаписью дерева снизу вверх с динамическим программированием для генерации ассемблерного кода, означающую, что сгенерированный код является локально (то есть, по выражениям) оптимальным, пока мы помещаем в него правильные описания машинного образа. Кроме того, выходной генератор может выполнять следующие оптимизации. Примечательно, что они являются расширениями ImageCraft и не являются частью стандартной дистрибуции LCC.

- Глазковая оптимизация.

  В то время как локально код может быть оптимальным, сгенерированный код может все еще иметь избыточные фрагменты, следующие из различий инструкций Си. Глазковая оптимизация устраняет некоторые из этих излишков.

- Распределение регистров.

  Для процессоров с многочисленными машинными регистрами (например, AVR, MSP430, и ARM), для каждой функции, компилятор выполняет распределение регистров и пробует упаковать так много локальных переменных насколько возможно в машинные регистры и тем самым увеличивает эффективность сгенерированного кода. Мы используем сложный алгоритм, который анализирует использование переменных (например, область программы, где они используются) и может даже помещать несколько переменных в одни и те же регистры, если их использование не накладывается.

- История регистров.

  Это работает в тандеме с распределением регистров и отслеживает содержание регистров и устраняет копии и другие подобные излишества.

1.12.1. Компрессор кода – Code Compressor™

Компрессор кода доступен в избранных трансляторах. Это работает после того, как вся программа скомпонована и заменяет повторно используемые фрагменты кода на обращения к функции. Это может уменьшать размер программы до 30 % (типичное значение – от 5 % до 15 %) без значительного замедления быстродействия. См. 12.1. Компрессор кода – Code Compressor (tm).

1.12.2. Машинно-независимый оптимизатор

MIO – Machine Independent Optimizer – передовой оптимизатор на уровне функций. Он выполняет следующие оптимизации на уровне функций, учитывая эффект структур потока управления:

- Распространение констант.

  Отлеживается назначение константы локальной переменной, и если возможно, использование переменной заменяется константой. В комбинации со сверткой константы, может быть очень эффективной оптимизацией.

- Глобальная нумерация значений.

  Подобно удалению общего подвыражения. Это удаляет избыточные выражения на уровне функции.

- Перемещение кода инварианта цикла.

  Выражения, которые не изменяют внутренние циклы, перемещаются наружу.

- Расширенное распределение регистров.

  Уже и так мощный распределитель регистров усилен “сетью” (не Internet web) формирующей процесс, который эффективно использует одиночную переменную как многократно используемую, позволяя выполнить лучшее распределение регистров.

- Усовершенствованное размещение локальных переменных стека.

  Даже с расширенным размещением регистров, иногда целевой процессор не имеет достаточно машинных регистров, чтобы хранить всех кандидатов в локальные переменные в регистрах. Эти дополнительные переменные нужно помещать в стек. Однако, большое смещение стека в общем случае менее эффективно в большинстве целевых процессоров. Используя тот же самый быстрый алгоритм, неразмещенные переменные оптимально упаковываются, используя память совместно, по возможности уменьшая полный размер кадра стека функции.

ImageCraft приложил значительные усилия при применении современной инфраструктуры оптимизатора. В настоящее время MIO оптимизация приносит пользу, главным образом улучшая быстродействие и несколько уменьшая размера кода. Мы продолжим совершенствовать оптимизатор и добавим новые виды оптимизации по мере развития системы.

Оптимизацию сжатия кода можно допускать в дополнение к MIO оптимизации. Эта комбинация дает вам минимальный код при некотором ухудшении производительности, вызываемой компрессором кода.

1.13. Благодарности

Входная часть компилятора Icc: "Icc source code (C) 1995, by David R. Hanson and AT&T. Воспроизводится с разрешения." Ассемблер/компоновщик – отдаленный потомок пакета ассемблера/компоновщика Alan Baldwin. Утилита Make – Jacob Navia. Попробуйте недорогой Win32 компилятор Jacob Navia на http://www.cs.virginia.edu/~lcc-win32.

УСОВЕРШЕНСТВОВАННЫЕ и ПРОФЕССИОНАЛЬНЫЕ версии включают утилиты GNU RCS и программу grep. GNU copyleft лицензия определяет, что вы можете распространять GNU программы. Это неприменимо к любому другому программному обеспечению в данном пакете, которое не основано на GNU. ImageCraft не модифицировал программы GNU. Исходные и двоичные коды GNU могут быть найдены на http://www.gnu.org.

Инсталляция использует программу 7 Zip (7za.exe) для распаковки некоторых из файлов. Копия программы установлена в `\iccv7430\bin`. 7 Zip использует GNU LGPL лицензию, и вы можете получить копию программы с сайта http://www.7-zip.org.

Весь код используется с разрешения. **Пожалуйста, сообщайте обо всех ошибках непосредственно нам.**

2. ВВЕДЕНИЕ

2.1. Быстрый старт

После запуска интегральной среды разработки (IDE), выберите *Project>Open...* из системы меню. Перейдите в каталог `\iccv7430\examples.430` и выберите проект `"led.prj"`. Менеджер проекта отобразит имя файла `led.c`, показывая, что в этом проекте только один файл. В опциях компиляции проекта *Project>Options...>* на вкладке **"Target"** выберите целевой процессор.

Теперь выберите *Project>Make Project*. Среда разработки вызовет компилятор для компиляции файлов проекта и выведет некоторые сообщения в окне состояния.

В случае отсутствия ошибок, в том же каталоге, где находится ваш исходный файл, будет создан выходной файл с именем `led.hex`. В данном случае в `\iccv7430\examples.430`. Этот файл создается в формате Intel HEX. Большинство программаторов и симуляторов MSP430 понимают этот формат, и вы можете загрузить эту программу в ваш контроллер. Это все, что нужно для создания данной программы. Если вы хотите отладить программу, используя опциональный отладчик NoICE430, вызовите его через панель инструментов или в меню *Tools>NoICE430...* Затем откройте отладочный файл в формате `.dbg` и вы получите возможность полной отладки на уровне исходного кода Си.

Обратите внимание, что часто используемые функции доступны также на панели инструментов и в контекстно-чувствительном всплывающем меню по щелчку правой кнопки мыши. Например, вы можете выбрать опции компилятора правым щелчком в окне проекта.

Двойной щелчок на имени файла в окне проекта открывает его в редакторе. Откройте `led.c` этим способом и для эксперимента, попробуйте создать ошибку удалением из строки точки с запятой. Теперь выберите *Project>Build*. Среда спросит вас о желании сохранить изменения. Выберите **"Yes"**, и начнется компиляция. На этот раз должна появиться ошибка, отображаемая в окне состояния. Щелчок на ошибочной строке или на символе ошибки слева от нее переместит курсор на ошибочную строку в редакторе.

2.1.1. Старт нового проекта

Введите команду *Project>New* и выберите каталог, в который вы хотите поместить ваши файлы проекта. Имя выходного файла основывается на имени вашего файла проекта. Например, если вы создаете проект с именем `foo.prj`, имя выходного файла будет `foo.hex` и т.п.

Создав ваш проект, вы можете начинать писать исходный текст (на Си или ассемблере) и добавлять исходные файлы в список файлов проекта. См. 2.4. Использование менеджера проекта. Компиляция проекта производится простым щелчком кнопки "**Build**" на панели инструментов.

Чтобы упростить процесс разработки, вы можете использовать средства Application Builder (См. 3.6. Application Builder) для создания кода инициализации периферии.

2.2. Анатомия Си программы

Программа на Си должна определять функцию с именем main. Компилятор связывает вашу программу со стартовым кодом и библиотечными функциями в "исполнимый" файл, называемый так потому, что его можно исполнить в целевом контроллере. Назначение стартового кода подробно описано в разделе 7.2. Файл запуска. Программа на Си нуждается в настройке целевой среды специальным образом, и стартовый код выполняет инициализацию целевого процессора для выполнения этих требований.

Обычно, main производит некоторую инициализацию и затем выполняет бесконечный цикл. Смотрите файл led_blinker.c в каталоге \iccv7430\examples:

```
#include <msp430x14x.h>
#define STOP_WATCHDOG WDTCTL = WDTPW + WDTHOLD;

int main() { // blink in a short-long-short pattern

  // slowdown factor
  #define SLOWDOWN_COUNT 5

  unsigned int i;
  unsigned int j;

  // avert Watchdog reset
  STOP_WATCHDOG

  // set LED pin as output
  P1DIR = BIT0;

  // infinite blink loop
  while(1) {

    // short ON
    P1OUT = BIT0;
    for(i = 0; i < 10000; i++){for(j = 0; j < SLOWDOWN_COUNT; j++);};

    // medium-length OFF
    P1OUT = 0;
    for(i = 0; i < 25000; i++){for(j = 0; j < SLOWDOWN_COUNT; j++);};

    // long ON
    P1OUT = BIT0;
    for(i = 0; i < 60000; i++){for(j = 0; j < SLOWDOWN_COUNT; j++);};

    // medium-length OFF
    P1OUT = 0;
    for(i = 0; i < 25000; i++){for(j = 0; j < SLOWDOWN_COUNT; j++);};

  } // infinite blink loop

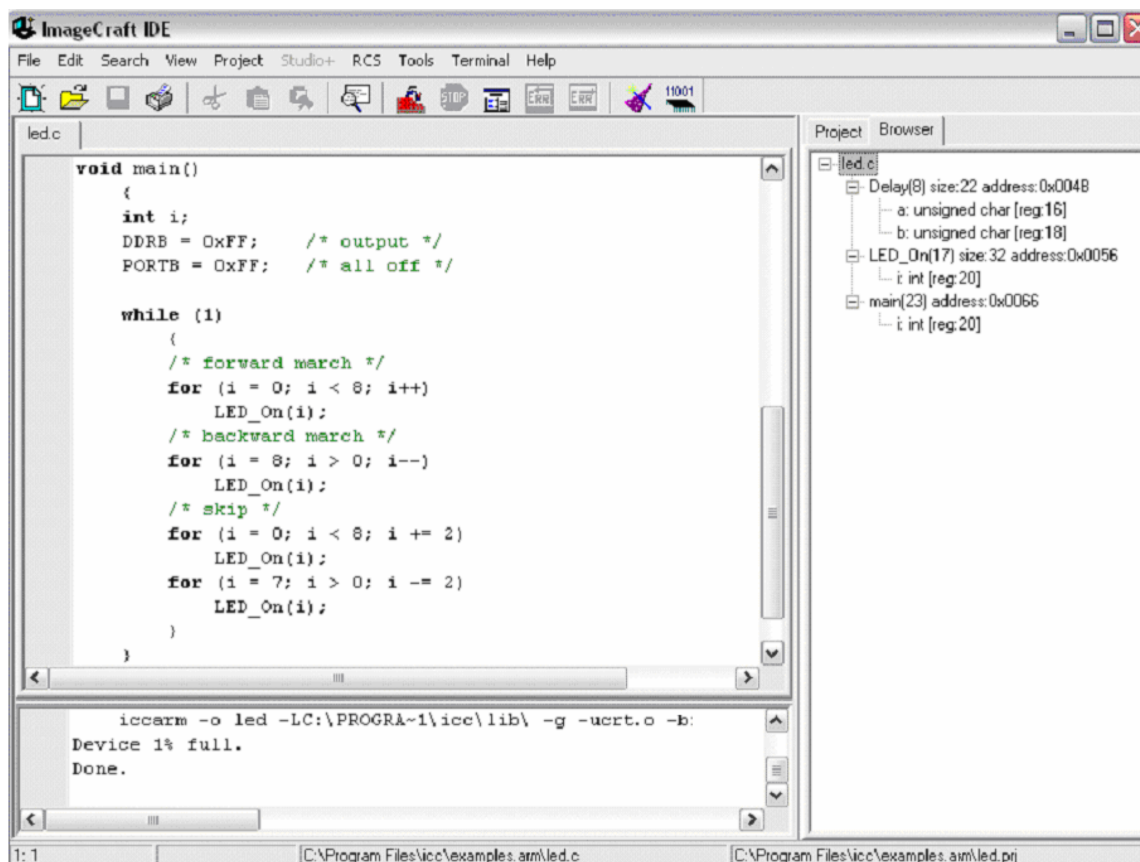
  return 0;

} // main
```

Программа очень проста. Она лишь устанавливает P1OUT в 0 или 1, включая и выключая светодиод. Циклы временной задержки используются для того, чтобы светодиод оставался включенным или выключенным на заметное время. Другие примеры находятся в папке /icc/examples.430, включая программу мигания светодиода, используя прерывания.

2.3. Краткий обзор среды разработки

Главное окно среды разработки IDE разделено на три части:



Левая верхняя часть окна – редактор. Он содержит окна редактируемых файлов и терминала. Редактор может осуществлять цветовую подсветку синтаксических конструкций Си, отображать закладки и другие элементы. При открытии эмулятора терминала, он отображается как одно из окон редактора. (См. 3.8. Эмулятор терминала)

Правая верхняя часть – окно менеджера проекта, содержащее две вкладки. Одна вкладка содержит список файлов проекта, другая – обозреватель кода, показывающий список функций и переменных, определенных в вашем проекте. См. 3.3. Список файлов проекта и окно обозревателя кода. При двойном щелчке на функции в обозревателе кода, курсор переместится на определение функции в исходном файле.

Нижняя часть – 3.7. Окно состояния. Здесь отображается состояние компиляции. Кроме того, строка состояния в нижней части отображает полезную информацию – полное имя файла в активном окне редактора, позиция курсора и полное имя файла проекта.

Окна могут изменять размер, и вы можете скрыть окно состояния или окно проекта, чтобы максимизировать окно редактора. Это выполняется установками в 4.5. Меню View.

Обычно операции пользователя вызываются при помощи меню. Часто используемые операции также доступны через кнопки на панели инструментов и через всплывающие контекстные меню при нажатии правой кнопки мыши. Среда гибко конфигурируется в меню 4.16. Опции редактора и печати. Вы можете переключаться между окнами редактора, нажимая на вкладки с именами файлов и окон или комбинацию клавиш <Ctrl-TAB>.

Среда включает 3.6. Application Builder, который создает код инициализации периферии выбранного устройства, упрощая начало написания ваших программ.

2.4. Использование менеджера проекта

Когда вы создаете ваши программные файлы во встроенном в среду или в каком-либо другом редакторе, вы можете добавлять файлы в менеджер проекта. Менеджер проекта отслеживает все файлы проекта, включая файлы, не содержащие исходный код, например – документацию проекта. Менеджеру проекта важны только файлы исходного кода. Когда вы выбираете команду **Build Project**, менеджер проекта определяет зависимости заголовочных файлов и вызывает компилятор, чтобы перекомпилировать только те файлы, которые были изменены. Использование менеджера проекта значительно упрощает задачу программирования.

Обычно, при создании проекта, для упрощения его сопровождения, вы разбиваете код на несколько исходных файлов. См. 3.2. Управление проектом. В качестве совета хорошего стиля программирования см. 6.1.3. Структура исходного текста и заголовочные файлы и т.п. Хоть это и не рекомендуется, но для быстрого и грубого макетирования, вы можете обойти шаги установки проекта, компилируя одиночный файл в выходной файл. См. 3.4. Компиляция отдельного файла.

3. СРЕДА РАЗРАБОТКИ

3.1. Концепция среды разработки

Среда разработки ImageCraft IDE разработана так, чтобы упростить ее использование. Вы организовываете ваши исходные файлы в проект, определяете параметры проекта (например, целевое устройство и другие опции компилятора) и вызываете команду меню *Project>Build* (или нажимаете на кнопку **"Build"** на панели инструментов) чтобы скомпилировать ваш проект всякий раз, когда вы изменяете исходные файлы. Имеются некоторые дополнительные средства, такие как окно просмотра кода (Code Browser), которое дает вам информацию о вашей программе, окно терминала для связи с устройствами по RS-232. В некоторых продуктах существует Application Builder для генерации кода инициализации периферии через графический интерфейс пользователя (GUI) и прямая поддержка программирования целевых устройств.

3.2. Управление проектом

Менеджер проекта среды разработки позволяет составлять проект из списка файлов. Это дает возможность разбивать программу на небольшие модули. При выполнении функции *Project>Build*, перекомпилируются только те исходные файлы, которые подверглись изменениям. При этом автоматически генерируются зависимости заголовочных файлов. Это значит, что если исходный файл включает заголовочный файл, то исходный файл будет автоматически перекомпилирован, если заголовочный файл изменился.

При компиляции проекта, менеджер проекта создает make-файл в стандартном формате. Сгенерированный make-файл можно исследовать при помощи команды *View>Project Makefile*.

3.2.1. Создание нового проекта

Чтобы создать новый проект, используется команда меню *Project>New*. Откроется диалоговое окно, позволяющее определить имя проекта, которое также используется как имя выходного файла. Если какие-либо исходные файлы уже созданы, их можно добавить в проект, используя команду *Project>Add File(s)...*. В противном случае, при помощи команды *File>New* можно создать новые исходные файлы, ввести в них исходный текст и затем сохранить их командой *File>Save* или *File>Save As...*. Сохраненные файлы можно затем добавить в проект при помощи команды *Project>Add File(s)...*. Допускается также добавлять в проект файл, который в настоящее время редактируется. Для этого можно щелкнуть правой кнопкой мыши в окне редактора и выбрать команду **"Add to Project"** во всплывающем меню. Обычно исходные файлы помещаются в тот же каталог, в котором находятся файлы проекта, однако это не является обязательным требованием.

Опции компилятора устанавливаются в меню *Project>Options*.

3.2.2. Опции проекта

Опции компилятора настраиваются в окне, вызываемом командой *Project>Options...*, и сохраняются вместе с файлами проекта, поэтому можно иметь различные проекты с различными целевыми контроллерами. Когда вы начинаете новый проект, используется заданный по умолчанию набор опций. Вы можете установить текущие опции как опции по умолчанию или загрузить опции по умолчанию в текущий набор опций. Опции по умолчанию сохраняются в файле `deficc430.prj` в каталоге размещения компилятора.

Чтобы избежать загромождения каталога с вашим проектом, вы можете определить, что выходные и промежуточные файлы, которые генерируются инструментальными средствами, будут находиться в отдельном каталоге. Обычно, это подкаталог в вашем каталоге проекта. См. 4.11. Опции компилятора: Пути.

3.2.3. Компиляция проекта

Вы компилируете проект, вызывая *Project>Build* или щелкая кнопку **"Build Project"**. Менеджер проекта перекомпилирует только те файлы, которые были изменены. Это может сэкономить значительное количество времени, когда ваш проект становится большим. В некоторых редких случаях, если менеджер проекта не перекомпилирует исходный файл, когда это необходимо, вы можете выполнить команду *Project>Rebuild All*, чтобы перекомпилировать все исходные файлы.

3.2.4. Перемещение проекта

Для перемещения проекта в другой каталог или на другую машину, в дополнение к вашим исходным файлам, вы должны переместить только `.prj` и `.src` файлы. Среда разработки не использует какие-либо другие скрытые файлы в вашем проекте. Если вы сохраняете ту же самую структуру папок, то более ничего не нужно выполнять. Файл `.prj` содержит установки среды проекта и не должен изменяться вручную. Файл `.src` содержит список файлов проекта. Это текстовый файл и, в некоторых редких случаях, вы можете редактировать `.src` файл вручную, чтобы обойти некоторые проблемы. Например, имена файлов проекта сохранены, используя относительные пути там, где это возможно. Если вы меняете пути ваших исходных файлов, вы можете отредактировать `.src` файл непосредственно, чтобы отразить новую структуру путей.

3.3. Список файлов проекта и окно обозревателя кода

Вы можете добавлять файлы в менеджер проекта, используя *Project>Add Files...* или, если файл открыт в редакторе, вы можете щелчком правой кнопки мыши вызвать всплывающее меню, чтобы выбрать "**Add to Project**". Файлы в списке менеджера проекта можно менять местами, цепляя и перемещая их мышью.

Исходный файл может быть написан или на Си или на ассемблере. Файлы на Си должны иметь расширение `.c` , а файлы на ассемблере должны иметь расширение `.s`. Вы можете хранить в списке файлов проекта любые файлы. Например, вы можете хранить файлы документации проекта в окне менеджера проекта. Менеджер проекта игнорирует файлы, отличные от файлов с исходным кодом, при выполнении компиляции.

3.3.1. Обозреватель кода

Окно обозревателя кода отображает адреса, типы функций, локальные и глобальные переменные вашего проекта. Эта панель автоматически обновляется всякий раз, когда вы компилируете ваш проект. В окне просмотра, двойной щелчок на имени функции переместит курсор в место определения функции в исходном файле.

Содержание окна обозревателя кода обычно автоматически сортируется по именам функций или именам файлов, в зависимости от установок в 4.14. Опции среды разработки. Если ваш проект содержит слишком много символов, сортировка не будет выполнена автоматически, и вы получите информационное сообщение, когда впервые компилируете ваш проект. Вы можете сортировать содержание вручную, вызывая команду меню *Project>Manual Sort Browser Window*.

3.4. Компиляция отдельного файла

Обычно вы создаете проект и определяете все исходные файлы, принадлежащие этому проекту, а затем компилируете проект, чтобы создать выходной файл. Однако иногда удобно компилировать одиночный файл в объектный файл или в конечный выходной файл. Вы можете использовать команду среды *File>Compile File...*, чтобы выполнить любую из этих задач. Когда вы вызываете эту команду, компилируется файл в текущем активном окне редактора.

Компиляция отдельного файла в объектный файл полезна для проверки на синтаксические ошибки или если вы компилируете новый файл запуска. Компиляция отдельного файла в выходной файл полезна, если ваша программа небольшая и может храниться в одном файле. Обратите внимание, что используются заданные по умолчанию опции компилятора.

3.5. Редактор

Окно редактора – основная область взаимодействия между вами и средой разработки. Оно содержит открытые файлы проекта в виде окон с вкладками. Если вы вызовете встроенный эмулятор терминала, он также откроется в этом месте. Вы можете переключаться между окнами редактора, щелкая на вкладке с именем файла или нажимая комбинацию клавиш <Ctrl-TAB>.

Редактор является гибко конфигурируемым инструментом. См. 4.14. Опции среды разработки и 4.14.2. Предпочтительный редактор. Вы должны выбрать – использовать ли встроенный в среду редактор или внешний редактор. Режимы встроенного редактора устанавливаются в опциях редактора и печати. Например, вы можете изменить назначения клавиш (копия, вставить, и т.д.) на какие-либо более вам знакомые. Если вы используете внешний редактор, необходимо определить имя и полный путь к используемому редактору. Также необходимо определить параметры команд для внешнего редактора. Если вы не хотите использовать встроенный в среду редактор, вы можете использовать редактор по вашему выбору. Формат выходных сообщений компилятора прост и должен быть распознаваем большинством современных редакторов. Сообщения об ошибках имеют следующий формат:

```
!E filename(lineno): ...
!W filename(lineno):[warning] ...
```

Свяжитесь с поставщиком редактора, если нуждаетесь в дополнительной справке. Позволяя параллельный просмотр и редактирование того же самого файла в редакторе среды и внешнем редакторе одновременно, вы можете заставить среду обнаруживать изменения в любых открытых файлах, если они были изменены на диске (например, внешним редактором) и перезагрузить измененный файл.

Слева в окне редактора находится желоб – вертикальная полоса для отображения информационных знаков. Они включают также номера строк и закладки. Вы можете установить до 10 закладок в каждом окне редактора.

3.5.1. Внешние редакторы

Среда разработки позволяет вам выбрать внешний редактор как редактор, заданный по умолчанию. См. 4.14. Опции среды разработки. Если выбран такой режим, и вы делаете двойной щелчок на строке с ошибкой или на имени файла в списке проекта, файл будет открыт во внешнем редакторе.

3.6. Application Builder

Application Builder – это встроенный инструмент для создания кода инициализации периферийных устройств. Он вызывается нажатием кнопки **"Wizard"** на панели инструментов или выбором пункта меню *Tools>Application Builder*.

Application Builder использует целевой процессор, который вы определяете в *Project>Options* как заданное по умолчанию устройство.

Application Builder отображает вкладки диалоговых панелей для каждой периферийной подсистемы целевого контроллера. Операции должны быть очевидны. Вы разрешаете периферийное устройство установкой флага “**Enable**”, и устанавливаете параметры периферийного устройства, выбирая отображаемые опции. Для программируемых цифровых портов ввода/вывода (IO), вы можете выбрать, будет ли вывод порта входным или выходным, нажимая на переключатель “**Direction**”. Символ "i" отображается в поле флажка, если это – входной вывод и "o", если это – выходной вывод. Если вывод – входной, вы можете затем переключить поле "**Value**” в значение "стрелка вверх" или пробел в зависимости от того, хотите ли вы, чтобы вывод использовал подтягивающий резистор, который будет активирован, или нет. Если вывод – выходной вывод, то вы можете переключить поле "**Value**", чтобы установить его в "1" или "0".

Если вы перемещаете курсор мыши через элемент контроллера, появляется всплывающая подсказка с описанием и именем соответствующего регистра ввода/вывода (IO), именем и номером внешнего вывода и т.д.

Вы можете исследовать сгенерированный код, нажав на кнопку **"Preview"**, а также сохранить этот код, нажав на кнопку **"Save as..."**. Когда вы будете удовлетворены вашим выбором, щелкните **"OK"**, Application Builder завершит работу и перешлет сгенерированный код в новое окно редактора. Нажатие **"Cancel"** завершает его работу без сохранения сгенерированного кода.

Чтобы использовать сгенерированный код, сохраните его в файле, включите файл в список файлов вашего проекта, и сделайте следующее в вашей функции `main()`:

```
extern void init_devices(void); // declare the function
...
init_devices();
```

В простейшем случае, вы можете установить переключатель **"Include main()**", и сгенерированный код будет включать функцию `main()`, которая вызывает функцию `init_devices`, и все, что вам нужно сделать затем, это поместить ваш код в функцию `main()`.

3.7. Окно состояния

Окно состояния отображает информацию состояния среды разработки, такую, как состояние компиляции. Сообщения об ошибках компиляции начинаются с `!E...` и отмечаются маленьким красным значком на желобе слева. Щелчок на строке ошибки перемещает курсор на строку, вызвавшую ошибку, в окне редактора.

Содержание окна состояния может быть выделено и скопировано в буфер обмена Windows. Когда вы выполняете компиляцию, последняя строка указывает состояние процесса. Если имеется какая-либо ошибка, вы можете пролистать окно назад, чтобы найти источник ошибки.

3.8. Эмулятор терминала

Среда разработки содержит простой встроенный эмулятор терминала. Он обеспечивает базовые функции таких программ и поддерживает escape-последовательности ANSI терминала. Вы можете использовать его для связи с вашим бортовым монитором в целевом микроконтроллере, или в вашем устройстве для отображения диагностических сообщений.

4. СИСТЕМА МЕНЮ

4.1. Всплывающие меню

Всплывающие контекстно-зависимые меню доступны в большинстве окон по щелчку правой кнопки мыши. Они обычно содержат подмножество наиболее популярных команд окна.

4.2. Меню File

Меню File содержит операции с файлами и программой. Активное окно редактора – это окно с вкладкой, находящееся в фокусе, т.е. вынесенное на самый верх. При попытке открытия уже открытого файла, редактор сделает активным его окно. Строка состояния в нижней части дает информацию об активном окне редактора, включая позицию курсора, открыт ли файл ТОЛЬКО ДЛЯ ЧТЕНИЯ (READ ONLY), изменен ли он и полное имя файла.

- **New** – создать вкладку с новым пустым окном редактора для ввода текста.
- **Reopen**... – открытие файла из списка недавно открывавшихся файлов.
- **Open...** – открыть существующий файл для редактирования.
- **Reload... from Disk** – отказаться от изменений и перезагрузить открытый файл с диска.
- **Reload... from Backup** – перезагрузить открытый файл из резервной копии. См. Save.
- **Save** – сохранить открытый файл на диске. Перед сохранением нового содержимого, опционально создается файл резервной копии `<file>.<~ext>`. См. 4.14. Опции среды разработки.
- **Save As...** – сохранить открытый файл на диске под новым именем.
- **Close** – закрыть активный редактируемый файл. Выводится запрос на запись несохраненных изменений.
- **Compile File... To Object** – компилировать открытый файл в объектный файл с расширением `.o`. Заметим, что объектный файл не является загружаемым в программатор или симулятор. См. 1.9. Типы и расширения файлов. Это полезно для проверки файла на ошибки, создания объектного файла для библиотеки, или чтобы создать новый файл запуска. См. 7.2. Файл запуска.
- **Compile File... To Output** – компилировать открытый файл в выходной, подходящий для загрузки в программатор или симулятор. Обычно, для управления файлами вашего проекта, используется 3.3. Список файлов проекта и окно обозревателя кода, но если проект небольшой, вы можете использовать эту команду, чтобы создать выходной файл. Используются заданные по умолчанию 4.10. Опции компилятора.
- **Compile File... Startup File to Object** – то же, что компиляция “Compile File... to Object”, за исключением того, что для ассемблирования устанавливается ключ `-n`. Вообще это используется только для ассемблирования файла запуска. Обычно ассемблер неявно вставляет `".area text"` в начало файла, который обрабатывает. Это позволяет в общем случае опускать директивы области в начале модуля кода для правильного ассемблирования. Однако файл запуска имеет специальные требования, чтобы эта неявная вставка не производилась.
- **Save All** – сохранить все открытые в настоящее время файлы.
- **Close All** – закрыть все открытые в настоящее время файлы и окно терминала, если оно открыто. Выводится запрос на запись несохраненных изменений.
- **Print** – печать активного файла. См. 4.16. Опции редактора и печати.
- **Exit** – выход из программы. Выводится запрос на запись несохраненных изменений.

4.3. Меню Edit

Это меню содержит команды редактирования.

- **Undo** – отменить последние изменения в окне редактирования.
- **Redo** – отменить последнюю "отмену". Повторно вводит изменения, которые вы отменили.
- **Cut** – вырезать выделенный текст в буфер обмена Windows.
- **Copy** – копировать выделенный текст в буфер обмена Windows. Обратите внимание, что это работает также для содержимого окна состояния.
- **Paste** – вставить содержимое буфера обмена Windows в позицию курсора.
- **Delete** – удалить выделенный текст.
- **Select All** – выделить все содержимое активного окна редактора.
- **Block Indent** – сдвинуть выделенный текст вправо на расстояние определенное в меню 4.14. Опции среды разработки.
- **Block Outdent** – сдвинуть выделенный текст влево на расстояние определенное в 4.14. Опции среды разработки.

4.4. Меню Search

Это меню содержит функции поиска в редакторе.

- **Find...** – поиск текста в окне редактора. Опции поиска:
  - **Match Case** – если флаг установлен, учитывается регистр символов.
  - **Whole Word** – если флаг установлен, соответствие находится только, если строка поиска окружена пробельными символами или символами пунктуации.
  - **Direction Up/Down** – выбор направления поиска, вверх или вниз от курсора.
- **Find in Files...** – поиск текста во всех открытых файлах, или во всех файлах проекта, или в файлах, соответствующих определенной маске (по умолчанию – *.* или все файлы). Результат поиска отображается в окне состояния. Опции поиска:
  - **Case Sensitive** – если флаг установлен, учитывается регистр символов.
  - **Whole Word** – если флаг установлен, соответствие находится только, если строка поиска окружена пробельными символами или знаками пунктуации.
  - **Regular Expression** – если флаг установлен, позволяет задавать регулярные выражения grep. Некоторые из наиболее часто используемых выражений:
    - . (точка) – любой символ
    - ^ – начало строки
    - $ – конец строки
    - [0-9] – любая цифра
    - [a-z] – любой символ нижнего регистра
    - (<expr1> | < expr2>) – соответствует выражению expr1 или expr2
    - ? – соответствует 0 или 1 разу появления предыдущего выражения
    - * – соответствует 0 или большему количеству появлений предыдущего выражения
- **Replace...** – заменить текст в окне редактора.
- **Find Again** – выполнить повторный поиск, используя последнюю строку поиска.
- **Jump to Matching Brace** – если курсор находится на символе скобки (то есть перед ним), курсор переместится вперед или назад к соответствующей парной скобке. Например, символ левой фигурной скобки соответствует символу правой фигурной скобки. Символы парных скобок: (,), [,], {,}, <, >.
- **Goto Line Number** – запрос номера строки и переход к ней. Обратите внимание, что вы можете также иметь редактор с номерами строк на полосе слева.
- **Goto First Error** – переход к строке с первой ошибкой в окне состояния.
- **Goto Next Error** – переход к строке со следующей ошибкой.
- **Add Bookmark** – установить на строке закладку. Обратите внимание, чтобы добавить или удалить закладку, часто быстрее просто щелкнуть на полосе слева от строки.
- **Delete Bookmark** – удалить закладку на строке.
- **Next Bookmark** – поиск вперед, пока не встретится строка с закладкой.
- **Goto Bookmark** – переход к указанной закладке.

4.5. Меню View

- **Project File List** – если флаг установлен, отображается окно списка файлов проекта (правая панель). Снимите отметку, чтобы максимизировать размеры окна редактора.
- **Status Window** – если флаг установлен, отображается окно состояния (нижняя панель). Снимите отметку, чтобы максимизировать размеры верхних панелей.
- **Project Makefile** – открыть make-файл в режиме ТОЛЬКО ДЛЯ ЧТЕНИЯ. Когда вы компилируете проект, автоматически создается make-файл, который описывает зависимости файлов проекта. Зависимости между заголовочными файлами (`.h` файлы) определяются средой автоматически.
- **Output Listing File** – открыть файл листинга (`.lst`) в режиме ТОЛЬКО ДЛЯ ЧТЕНИЯ. Листинг файл содержит заключительные адреса всего вашего программного кода, за исключением библиотечных процедур. См. 10.1.2. Файл листинга.

4.6. Меню Project

Меню содержит интерфейс с построителем проекта – Project Builder. Только один проект может быть открыт одновременно. Если имеется открытый проект с несохраненными изменениями, и вы попробуете создать новый проект или открыть другой проект, вам будет выдан запрос на сохранение изменений.

- **New...** – создать новый файл проекта. Выводится приглашение ввести каталог и имя файла, чтобы сохранить файл проекта. Обычно вы храните ваш файл проекта в каталоге с вашими исходными файлами, хотя это не обязательно. Имя, которое вы даете проекту, будет использоваться, как имя выходного файла. Например, если ваш проект назван `foo.prj`, то выходной файл будет называться `foo.hex`, `foo.cof` и т.д. в зависимости от формата выходного файла.
- **Open** – открыть существующий файл проекта.
- **Open All Files...** – открыть все исходные файлы проекта.
- **Close All Files** – закрыть все открытые файлы проекта.
- **Reopen...** – содержит список недавно открытых проектов. Выберите один проект для повторного открытия.
- **Make Project** – определяет зависимости файлов проекта и компилирует измененные файлы в выходной файл.
- **Rebuild All** – перекомпилировать все файлы проекта. Полезно, если что-либо не работает, как надо или, если вы находите, что ваши файлы не перекомпилируются, даже если они изменены, что может быть результатом неправильной системной даты.
- **Add File(s)** – открыть диалоговое окно для добавления файлов в проект. Вы можете добавлять в ваш проект любые файлы, но исходными файлами должны быть файлы Си (с расширением `.c`) или файлы ассемблера (с расширением `.s`). Файлы, не являющиеся исходным кодом продолжают список файлов проекта, но игнорируются Project Builder.
- **Remove Selected Files** – удалить из проекта файлы, выбранные в окне списка файлов проекта.
- **Option...** – открыть диалоговое окно Compiler Options. См. 4.10. Опции компилятора.
- **Manual Sort Browser Window** – обычно содержание окна просмотра кода автоматически сортируется согласно набору опций в 4.14. Опции среды разработки. Однако, если в окне просмотра кода находится слишком много элементов, они не будут сортироваться автоматически. Вы можете вызвать сортировку принудительно, используя эту команду.
- **Close** – закрыть проект. Запрашивается сохранение изменений, если необходимо.
- **Save** – сохранить проект, включая список файлов проекта и опций компилятора.
- **Save As...** – сохранить проект под другим именем.

4.7. Меню RCS

Общий обзор RCS см. в 12.2. Система управления версиями.

4.7.1. RCS функции среды разработки

- **Checkin Selected File(s)** – проверка всех выбранных файлов в окне списка проекта. Отображается диалоговое для того, чтобы ввести запись регистрации (или, в случае начальной регистрации, описание файла и опциональную метку). Файлы будут проверены немедленно после этого. Для проверки файлов используется команда `ci` с опцией `-l`.
- **Checkin Project** – проверка всех файлов проекта. Вы получите сообщение об ошибке от `ci`, если файл не был изменен; вы можете игнорировать эти ошибки. Файлы будут проверены немедленно после этого.
- **Diff Selected File** – отображает отличия между версиями файла. По умолчанию должны сравниваться последняя версия с версией, с которой вы в настоящее время работаете. Вы можете также сравнивать любые две версии файла. Результат отображается в окне состояния. Для этого используется команда `rcsdiff`.
- **Show Log of the Selected File(s)** – показывает записи регистрации выбранных файлов. Используется для нахождения различных номеров версий и меток файлов. Для этого используется команда `rlog`.

4.8. Меню Tools

- **Environment Options** – открыть диалоговое окно 4.14. Опции среды разработки и 4.15. Опции терминала.
- **Editor and Print Options** – открыть диалоговое окно 4.16. Опции редактора и печати.
- **Application Builder** – вызов Application Builder, который является утилитой, создающей процедуры инициализации устройства, основанные на выборе опций, которые вы делаете в ряде диалоговых окон. Когда вы нажимаете **"OK"**, будет создано новое окно редактора, содержащее сгенерированный код. Application Builder имеет следующие кнопки управления:
  - **OK** – выход из Application Builder и помещение сгенерированного текста в новое окно редактора.
  - **Options...** – всплывающее меню, разрешающее сохранять и загружать установки Application Builder в файле конфигурации. Также позволяет включать сгенерированный текст в пустую функцию `main()` для получения полного скелета программы.
  - **Save As...** – сохранить сгенерированный код в файле.
  - **Preview** – предварительный просмотр сгенерированного кода во всплывающем окне.
  - **Cancel** – выход из Application Builder без сохранения сгенерированного текста.
- **Configure Tools** – позволяет добавлять инструментальные средства в меню Tools. При вызове вы можете использовать диалоговое окно, чтобы изменять содержание меню Tools. Поля этого диалогового окна:
  - **Menu Name** – имя для использования в меню Tools.
  - **Program** – определить полный путь к выполнимой программе. Вы можете использовать кнопку **"Browse"**, чтобы выбрать выполнимую программу.
  - **Parameters** – определить параметры программы. Распознается следующий формат параметров: `%f` заменяется именем верхнего редактируемого файла; `%p` заменяется именем текущего проекта; `%o` – имя выходного файла; и `%P` – имя файла проекта с полным выходным путем.
  - **Initial Directory** – каталог, в который среда переключается перед выполнением программы.
  - **Capture Output** – если флаг установлен, среда перехватывает вывод программы (стандартный вывод и стандартная ошибка) и показывает его в окне состояния. Это должно использоваться только с консольными Win32 или DOS приложениями.
- **Run** – простой интерфейс для выполнения программ из командной строки. Подобен Windows функции "Run" в меню "Старт". Любые инструментальные средства, которые вы конфигурируете, выполняются при помощи команды "Run".

4.9. Меню Terminal

Это меню взаимодействует с эмулятором терминала.

- **View** – ключ отображения эмулятора терминала.
- **Clear Window** – очистка окна терминала.
- **Capture...** – переключатель перехвата вывода терминала. Предлагается ввести имя файла, когда включено.

4.10. Опции компилятора

В диалоговом окне опций имеются три вкладки: **"Paths"**, **"Compiler"** и **"Target"** – Пути, Компилятор и Целевое устройство. В дополнение к стандартным кнопкам **"OK"**, **"Cancel"** и **"Help"**, имеются еще две:

- **Set As Default** – записать опции в файл опций по умолчанию `\iccv7430\bin\deficc430.prj`. Когда вы запускаете среду разработки или создаете новый проект, они загружаются как опции по умолчанию.
- **Load Default** – загрузить опции по умолчанию в текущий набор установок.

4.11. Опции компилятора: Пути

- **Include Paths** – определение каталогов, где компилятор должен искать файлы для включения. Вы можете определить несколько каталогов, разделяя пути точкой с запятой или пробелом. Если путь содержит пробел, то заключите его в двойные кавычки.

  Если вы определяете неполный путь (то есть путь, который не начинается с "\" или с буквы диска), то он определяется относительно выходного каталога – "Output Directory" (см. ниже) а не каталога проекта. Драйвер компилятора автоматически добавляет `\iccv7430\include` (замените `\iccv7430` на ваш путь установки) к путям включаемых файлов и вы не должны добавлять его явно.

- **Assembler Include Paths** – определение каталогов, где ассемблер должен искать файлы для включения. Вы можете определить несколько каталогов, разделяя пути точкой с запятой или пробелом. Если путь содержит пробел, то заключите его в двойные кавычки.

- **Library Paths** – определение каталогов, где компоновщик должен искать библиотечные файлы. Вы можете определить несколько каталогов, разделяя пути точкой с запятой или пробелом. Если путь содержит пробел, то заключите его в двойные кавычки. Драйвер компилятора автоматически добавляет `\iccv7430\lib` (замените `\iccv7430` на ваш путь установки) к путям библиотечных файлов, так что вы не должны добавлять его явно.

  Компилятор автоматически связывает заданную по умолчанию библиотеку Си и файл запуска (см. 7.2. Файл запуска) с вашей программой. Заданная по умолчанию библиотека Си находится в файле `libc430.a`. Файл Запуска `crt*.o` и библиотечные файлы должны быть размещены в каталогах библиотек.

- **Output Directory** – обычно исходные файлы хранятся в одном каталоге вместе с файлами проекта. Компиляция создает много файлов и чтобы избежать загромождения каталога проекта, вы можете поместить все выходные файлы в их собственный каталог. Обычно это – подкаталог в каталоге проекта.

4.12. Опции компилятора: Компилятор

- **Strict ANSI C Checking** – проверка на строгое соответствие стандарту ANSI. Стандарт ANSI позволяет некоторые операции, которые могут быть небезопасными. Если флаг активен, компилятор предупреждает об объявлениях функций без прототипов, присваиваниях между указателями на целые и перечислимые типы, о преобразованиях указателей в меньшие интегральные типы. Также предупреждается о нераспознанных управляющих строках, не ANSI расширениях языка, неразрешенных ссылках на статические переменные и функции, о массивах незавершенных типов, превышениях ограничений ANSI, таких, как число `case` более 257 в операторах `switch`.
- **Accept Extensions** – разрешить комментарии в стиле C++ и поддержку синтаксиса двоичных констант (например, `0b10101`).
- **Macro Define(s)** – определение макросов, отделяемых пробелами или точкой с запятой. Каждое макроопределение должно быть в форме:

  `name[:value]` *или* `name[=value]`

  Например:

  `DEBUG=1; PRINT=printf`

  определяет два макроса, `DEBUG` и `PRINT`. `DEBUG` имеет значение 1 по умолчанию, а `PRINT` определен как `printf`. Это эквивалентно записи

  ```
  #define DEBUG 1
  #define PRINT printf
  ```

  в исходном тексте. Общее применение состоит в использовании условных директив препроцессора для включения или исключения некоторых кодовых фрагментов.
- **Macro Undefine(s)** – тот же синтаксис, что в Macro Define(s), но отменяет макрос.
- **Output File Format** – выбор формата выходного файла. Обычно программатор требует файл Intel HEX или Motorola S19. Если необходима отладка, выберите формат, включающий отладочную информацию.
- **Execute Command After Successful Build** – добавляет к сгенерированному make-файлу выполнение определенной пользователем команды после того, как проект успешно скомпилирован. Поддерживаются следующие `%<c>` символы:
  - `%p` – расширяется до имени проекта.
  - `%f` – расширяется до имени файла в текущем активном окне редактора.
  - `%o` – расширяется до пути выходного каталога.
  - `%P` – расширяется до имени проекта в выходном каталоге.
  - `%%` – расширяется до одиночного знака %.

4.13. Опции компилятора: Целевое устройство

- **Device Configuration** – выбор целевого микроконтроллера. Прежде всего, влияет на адреса, которые компоновщик использует при связывании ваших программ. Если ваше устройство отсутствует в списке, выберите "**Custom**" и введите подходящие параметры, описанные ниже. Если ваше устройство подобно существующему устройству, то сначала выберите подобное устройство и затем включите "**Custom**".
- **Code Start Address** – стартовый адрес FLASH памяти. Она расположена от этого адреса до 0xFFFF. Область от 0xFFDF до 0xFFFF содержит векторы прерываний.
- **RAM Size** – объем внутренней памяти данных RAM. Почти все MSP430 имеют память RAM с адреса 0x200. Некоторые устройства имеют память RAM с адреса 0x1100 (например, F1610). Среда автоматически устанавливает адрес RAM по типу устройства.
- **HW Multiply** – определяет наличие в устройстве аппаратного умножителя. Изменяется только при выборе пользовательского устройства – "Custom".
- **PRINTF Version** – эта группа переключателей позволяет вам выбирать, с какой версией `printf` будет связана ваша программа. Больше возможностей потребует большее количество кода. См. 7.6. Стандартные функции ввода/вывода для подробностей:
  - Малая или базовая версия: доступны только `%c, %d, %x, %X, %u,` и `%s` форматы без модификаторов.
  - Длинная: допускается длинный модификатор. В дополнение к полям ширины и точности, поддерживаются форматы `%ld, %lu, %lx`, и `%lX`.
  - С плавающей запятой: добавляются форматы `%e, %f` и `%g`.
- **Unused ROM Fill Bytes** – заполняет неиспользуемые области ROM определенным целочисленным шаблоном.
- **Additional Libraries** – использование других библиотек кроме стандартных, поддерживаемых программой. Например, на нашем веб-сайте есть библиотека по имени `libstk.a` для доступа к периферийным устройствам STK-200. Чтобы использовать другие библиотеки, скопируйте файлы в один из библиотечных каталогов и определите имена библиотечных файлов без префикса `lib` и расширения `.a` в этом поле. Например, `stk` относится к библиотечному файлу `libstk.a`. Все библиотечные файлы должны заканчиваться расширением .a.
- **Non Default Startup** – файл запуска всегда компонуется с вашей программой (см. 7.2. Файл запуска). В некоторых случаях, вы можете иметь другие файлы запуска на основе свойств проекта. Эта опция позволяет определять имя файла запуска. Если имя файла не имеет полного пути, то файл должен быть в одном из библиотечных каталогов.
- **Other Options** – позволяет вводить любые параметры командной строки компоновщика. Например, имеется исходный файл:

  ```
  #pragma text:bootloader
  void boot() ... // function definition
  #pragma text:text // reset
  ```

  В “Other Options” введите:

  ```
  -bbootloader:0x????
  ```

  где `0x????` является начальным адресом области "bootloader". Используется для размещения основной программы с начальным загрузчиком в верхней памяти. Если вы пишете автономный начальный загрузчик, то вы можете просто выбирать конфигурацию памяти "bootloader" в диалоговом окне *Project>Options>Target*.

4.14. Опции среды разработки

Это диалоговое окно управляет общими установками среды разработки:

- **Beep on Completing Build** – выдавать звуковой сигнал при завершении компиляции.
- **Verbose Compiler Output** – заставляет драйвер компилятора распечатывать каждый проход обработки файла. Это показывает точные ключи командной строки, переданные каждому проходу компилятора.
- **Multiple Row Editor Tabs** – изменить вид вкладок окон редактора, чтобы использовать несколько строк вкладок вместо одной строки со стрелкой прокрутки, когда число вкладок становится слишком большим.
- **Auto Save Files Before Compiling** – автоматически сохранять файлы проекта при запросе компиляции. Обычно запрашивается сохранение изменений для каждого измененного файла.
- **Create Backup on Save** – при сохранении файла, копировать последнюю версию в файл с именем `<file>.<~ext>` перед перезаписью файла с последними модификациями.
- **Undo Across Save** – позволить отмену изменений, даже если файл был сохранен.
- **Scan for Changes in Opened Files** – периодически просматривать открытые файлы, чтобы заметить изменения в дисковой версии. Это полезно, если вы используете внешний редактор и держите файл открытым также и в среде разработки.
- **Close Files on Project Close** – автоматически закрывать все файлы проекта при закрытии самого проекта.
- **Printer Setup** – вызов диалогового окна установок принтера.

4.14.1. Опции просмотра обозревателя кода

Определение опций сортировки содержимого в окне 3.3.1. Обозреватель кода, когда оно обновляется после компиляции проекта:

- **Unsorted** – не сортировать содержимое. Это может сэкономить некоторое время на медленных машинах, если число символов большое.
- **Sort Functions Alphabetically** – отображать функции в алфавитном порядке, с последующими глобальными переменными.
- **Sort Functions by File Names** – отображать функции по именам файлов в алфавитном порядке. Каждый файл содержит функции и переменные, определенные в файле.

4.14.2. Предпочтительный редактор

Вы можете выбрать, использовать ли встроенный в среду или внешний редактор. Опции встроенного редактора устанавливаются в 4.16. Опции редактора и печати. Если вы выбираете внешний редактор, вы должны определить имя и полный путь к выполнимой программе редактора. Вам также необходимо определить параметры команд для следующих функций:

- Открытие файла для редактирования. Вы должны определить эту информацию.
- Открытие файла только для чтения. Это полезно для открытия файлов, которые не предназначены для редактирования вручную, например *View>Makefile*.
- Открытие файла и переход к указанной строке. Полезно для перехода к строке ошибки.

В окнах редактирования параметров команд, вы можете использовать `%f`, чтобы сослаться на имя файла и `%l` для ссылки на номер строки. Информация внешнего редактора сохраняется в файле `\iccv7430\bin\editors.ini`. Мы обеспечиваем информацию для некоторых из наиболее популярных редакторов в файле `\iccv7430\bin\editors.installed`. Когда вы устанавливаете программу в первый раз, среда копирует `editors.installed` в `editors.ini`. Когда вы обновляете программу впоследствии, файл `editors.ini` остается нетронутым, так что ваши изменения сохраняются. После обновления, вы можете открыть файл `editors.installed` и копировать и вставлять информацию любого нового редактора в файл `editors.ini`.

Введите новый редактор, выбирая строку “---NEW---”. Вводите запрошенную информацию и щелкните на кнопке **"Add"**. Если вы выбираете существующий редактор и делаете любые изменения, например, добавляете компонент пути к предопределенному редактору, нажмите на кнопку **"Change"**, чтобы сделать изменения постоянными. Вы можете удалить из списка выбранный редактор, нажимая на кнопку **"Delete"**. Для любой из этих кнопок не имеется никаких действий отмены.

4.15. Опции терминала

Изменение номера COM-порта или скорости обмена в бодах закрывает COM-порт и вновь открывает его с новыми параметрами, если уже открыт.

- **COM Port** – определить COM-порт, который должен использовать эмулятор терминала.
- **Baudrate** – определить скорость обмена данными в бодах. Поддерживаются все стандартные скорости обмена Windows.
- **Flow Control** – определение метода управления потоком данных.

4.16. Опции редактора и печати

Имеются три вкладки, которые управляют операциями редактора и печати. Опции редактора, относящиеся к файлам (например, создавать ли резервную копию файла) являются частью раздела 4.14. Опции среды разработки. Некоторые опции описаны как неиспользуемые и игнорируются.

4.16.1. Опции

Печать

- **Wrap long lines** – переносить по словам длинные строки при печати.
- **Line numbers** – печатать номера строк.
- **Title in header** – печатать имя файла в заголовке.
- **Date in header** – печатать текущую дату и время в заголовке.
- **Page numbers** – печатать номера страниц.

Общее

- **Word wrap** – автоматически переносить по словам длинные строки на дисплее.
- **Override wrapping** – не используется и игнорируется.
- **Auto indent** – когда отключен режим автоматического переноса по словам, отступ символа выравнивается по первому не пробельному символу в предыдущей строке.
- **Smart TAB** – когда отключен режим автоматического переноса слов, клавиша <TAB> перемещает курсор к следующему не пробельному символу в предыдущей строке.
- **Smart fill** – если включено использование символа табуляции (см. ниже), эта опция заставляет редактор использовать минимальное число символов, составленное из пробелов и символов табуляции, чтобы заполнить требуемый промежуток. Иначе, используются только пробелы.
- **Use TAB character** – вставлять символ табуляции в текст. Если этот флаг не установлен, то вместо символа табуляции вставляется соответствующее число пробелов.
- **Line numbers in gutter** – показывать номера строк на левом вертикальном желобе.
- **Mark wrapped lines** – показывать черный треугольник на полосе для перенесенных строк.
- **Title as filename** – не используется и игнорируется.
- **Block cursor for overwrite** – показывать блочный курсор, когда редактор в режиме замены.
- **Word select** – двойной щелчок помечает слово, ближайшее к позиции курсора мыши.
- **Syntax highlight** – включить синтаксическую подсветку текста.
- **Cursor beyond EOL** – позволить курсору перемещаться за символ конца строки.
- **Show all chars** – показывать скрытые пробельные символы (применяется к символам табуляции, пробела, новой строки и перенесенным строкам).

Разное

- **Right Margin** – отображать правый отступ (серая вертикальная линия) в определенном столбце.
- **Gutter** – отображать желоб определенного размера. Обратите внимание, что желоб используется для отображения закладок, номеров строк и других элементов.
- **Block indent step size** – число шагов при использовании команд выравнивания Block Indent и Outdent.
- **TAB Columns** – определяет позиции столбцов табуляции. Если не определено, то для вычисления позиций табуляции используется значение “TAB stop”.
- **TAB stop** – число символов, используемое символом TAB, если не используется “TAB Columns".

4.16.2. Контекстная подсветка

Эта страница позволяет вам управлять контекстной подсветкой синтаксических конструкций в тексте исходного кода.

4.16.3. Назначения клавиш

Страница позволяет вам изменять назначения клавиш для команд редактирования.

4.16.4. Шаблоны кода

Эта страница позволяет определять и редактировать "шаблоны кода" – “code templates”, к которым можно обращаться с помощью комбинации горячих клавиш (<Ctrl-J> по умолчанию). Шаблоны кода полезны для вставки базовых синтаксических конструкций без необходимости их полного набора на клавиатуре. Поддерживается набор шаблонов для часто используемых элементов, таких, как управляющие структуры языка Си.

5. ПРЕПРОЦЕССОР Си

5.1. Диалекты препроцессора Си

Заданный по умолчанию препроцессор Си – препроцессор стандарта C86/C89.

5.2. Предопределенные макросы

Препроцессор поддерживает следующие предопределенные макросы: `__FILE__`, `__DATE__` и `__TIME__` расширяются в строковые литералы; `__LINE__` расширяется в целое число; `__STDC__` расширяется в константу 1.

Кроме того, драйвер предопределяет идентификатор `__IMAGECRAFT__` и макросы, специфичные для целевых устройств и программных продуктов:

Компилятор	Предопределенный макрос
ICCV7 for AVR	_AVR
ICCV7 for 430	_MSP430
ICC08	_HC08
ICC11	_HC11
ICC12	_HC12
ICCV7 for ARM	_ARM
ICCM8C	_M8C

Среда разработки также предопределяет идентификаторы, используемые в диалоговом окне списка устройств (см. 4.13. Опции компилятора: Целевое устройство). Например, "ATMEGA128" определено, когда это устройство выбрано как целевое устройство. Это позволяет писать условный код, основанный на типе устройства.

5.3. Поддерживаемые директивы

Длинные определения могут быть разбиты на отдельные строки, используя для конкатенации строк символ наклонной черты влево (backslash) в конце незаконченной строки.

5.3.1. Макроопределения

- `#define macname definition`

  Простое макроопределение. Все ссылки на `macname` будут заменены его определением `definition`.

- `#define macname(arg [,args]) definition`

  Функция-подобный макрос, позволяющий передавать параметры в макроопределение.

- `#undef macname`

  Отменяет определение `macname` как макрос. Используется для переопределения `macname` другим значением.

Стандарт C99 допускает в функция-подобном макроопределении переменное число аргументов.

5.3.2. Условная обработка

В директивах условной компиляции (`#if/#ifdef/#elif/#else/#endif`), группа строк исходного кода относится к строкам между текущей директивой и следующей директивой условной компиляции. Условные директивы должны быть корректно скомбинированы, например `#else`, если существует, должна быть последней директивой цепочки перед `#endif`. Последовательность условных директив формирует группу. Группы условных директив могут быть вложены.

- `defined(name)`

  Может использоваться только внутри выражения `#if`. Вычисляется в 1, если `name` – имя существующего макроса и в 0 в противном случае.

- `#if <expr>`

  Условная компиляция группы строк, если результат вычисления `<expr>` ненулевой. `<expr>` может содержать арифметические и логические операторы и `defined(name)`. Однако так как препроцессор отделен от соответствующего компилятора Си, выражения не могут содержать операторы `sizeof` или `typecast`.

- `#ifdef name / #ifndef name`

  Сокращения для `#if defined(name)` и `#if !defined(name)`, соответственно.

- `#elif <expr>`

  Если результат вычислений предыдущих условий нулевой, а выражения `<expr>` ненулевой, то компилируется группа строк, следующая за `#elif`.

- `#else`

  Если результат вычислений всех предыдущих условий нулевой, группа строк, следующая за `#else`, компилируется до `#endif`.

- `#endif`

  Заканчивает условную компиляцию группы строк.

5.3.3. Дополнительно

- `#include <file>` или `#include "file"`

  Обрабатывается содержимое указанного файла.

- `#line <line> [<"file">]`

  Устанавливает номер строки и, может быть, имя исходного файла.

- `#error "message"`

  Выводит сообщение об ошибке.

- `#warning "message"`

  Выводит предупреждающее сообщение. Является расширением ImageCraft.

- `#pragma ...`

  `#pragma` содержит специфические для компилятора расширения. См. 1.10. Прагмы и расширения.

5.4. Строковые литералы и склейка лексем

Знак #, предшествующий макропараметру в макроопределении создает строковый литерал. Например:

```
#define str(x) #x
```

Вызов `str(hello)` расширяется до символьной строки `"hello"`. Это особенно полезно в некоторых командах встроенного ассемблера `asm`. Препроцессор Си не расширяет макроимена внутри строки. Так, следующий пример не будет работать:

```
#define PORTB 5
...
asm("in R0,PORTB"); // will not work like wanted
```

Намерения программиста состояли в том, чтобы расширить `PORTB` внутри строки до `"5"`, но это не будет работать. При создании строкового литерала, это может быть выполнено следующим образом:

```
#define PORTB 5
#define str(x) #x
#define strx(x) str(x)
...
asm("in R0," strx(PORTB)); // expanded in asm("in R0,5");
```

Если два строковых литерала появляются вместе, компилятор Си обрабатывает их как одну строку.

Если две лексемы препроцессора отделяются знаком `##`, то препроцессор создает из них одиночную лексему. Например:

```
foo ## bar
```

обрабатывается так же, как если бы вы написали одиночную лексему `foobar`.

6. КРАТКОЕ ОПИСАНИЕ Си

6.1. Введение

По языку Си имеется много хороших учебников и учебных веб-сайтов. Ссылки на некоторые веб-сайты находятся по адресу: http://www.imagecraft.com/software.

Нажмите на сайте кнопку **"Resources"** и воспользуйтесь ссылками на различные учебные веб-сайты. Этот раздел дает очень краткое введение в Си, используя наши инструментальные средства компиляции. Некоторые практические советы позволят вам повысить эффективность ваших работ. Содержание этой главы основано на нашем мнении, но очевидно, что есть много других полезных идей и практического опыта. И конечно, это не заменяет хороший учебник или справочник.

6.1.1. Стандарты Си

Си “вышел” в мир коммерции из Bell Laboratories в конце 1970-х годов. К началу 1980-х годов было много компиляторов Си для больших ЭВМ, PC и даже для встроенных процессоров (чем больше вещи изменяются, тем больше они остаются самими собой...). Первый комитет по стандартизации языка Си ставил одну из основных целей – “формализовать имеющийся опыт в максимально возможной степени”. Поэтому первый стандарт языка (C86) работает в основном так же, как люди его использовали на практике, с добавлением только нескольких ключевых слов (`const` и `volatile`). Здесь помогает относительная простота Си – даже если вы сталкиваетесь с некоторыми проблемами совместимости, чтобы удовлетворить другому стандарту часто достаточно небольшой модификации программы.

Когда ISO поставил задачу стандартизации Си для международного сообщества, C86, вообще говоря, был принят с некоторыми незначительными изменениями и стал известным как C89. Это основные диалекты, которым компиляторы ImageCraft более или менее соответствуют. “Более или менее”, потому что имеются некоторые небольшие отличия (например, для всех процессоров кроме ARM, мы не поддерживаем 64-х разрядную арифметику с плавающей точкой двойной точности, а поддерживаем только 32-х разрядную). Однако в 99% случаев, при следовании стандарту языка C86/C89, наши компиляторы обеспечивают совместимость.

C99 – последний стандарт Си. Пока некоторые стремились к созданию нового Си как подмножества C++, здравомыслие преобладало и C99 выглядит замечательным подобием C89 с добавлением нескольких новых ключевых слов и типов данных (например, `_bool`, `complex`, `long long`, `long double` и т.п.). Мы обеспечим поддержку C99 в будущем.

Заглядывая вперед, EC++ (Embedded C++) – очень полезное подмножество C++ для встроенных систем. Он обладает большинством из свойств объектно-ориентированных языков (класс, перегрузка, и т.д.), но без некоторых bloat (шаблонов). Начиная с середины 1980-х, “стандартный” C++ был объектом почти ежемесячных изменений. После долгих ожиданий, язык, наконец, стабилизировался, и мы поддержим EC++ в будущем для отдельных процессоров типа ARM.

6.1.2. Порядок трансляции и препроцессор Си

Компилятор Си состоит из нескольких программ, которые преобразуют исходные файлы Си из одного формата в другой. Сначала препроцессор Си производит макрорасширения (например, `#define`), текстовые включения (например, `#include`) и т.п. в исходных файлах. Затем соответствующий компилятор транслирует файл в ассемблерный код, который затем обрабатывается ассемблером. Ассемблер транслирует файл в объектный код. В заключение, компоновщик собирает все объектные файлы и связывает их в законченную программу.

Имеется два наблюдения относительно этого процесса. Первое: препроцессор Си отделен от соответствующего компилятора и осуществляет только текстовую обработку. Имеются замечания относительно макроса `#define`, являющиеся следствием этого. Например, в макроопределении макропараметры желательно поместить в скобки, чтобы предотвратить неожиданные результаты:

```
#define mul1(a, b) a * b         // bad practice
#define mul2(a, b) (a) * (b)     // good practice

mul1(i + j, k);
mul2(i + j, k);
```

`mul1` дает неожиданный результат для параметров, в то время как `mul2` дает ожидаемый результат (конечно, использование `#define` для простых операций типа одиночного умножения не является хорошей идеей, но это – другая тема). Второе наблюдение: файлы Си транслируются в файлы ассемблера и затем обрабатываются ассемблером. Фактически, Си иногда называют ассемблером высокого уровня, так как объем трансляции между Си и ассемблером относительно невелик по сравнению с более сложными языками типа C++, Java, FORTRAN и т.д.

6.1.3. Структура исходного текста и заголовочные файлы

Ваша программа должна содержать функцию с именем `main`. Хорошая практика – разбить программу на отдельные исходные файлы, содержащие функционально связанные процедуры и данные. Кроме того, имея модульную структуру программы, перестроить проект можно гораздо быстрее, имея множество небольших файлов, чем один большой файл. При использовании среды разработки, вы добавляете каждый файл в проект, используя 3.3. Список файлов проекта и окно обозревателя кода. Для упрощения сопровождения программы, вы можете использовать 3.3.1. Обозреватель кода и другие инструменты, чтобы размещать функции и данные во множестве исходных файлов. Обратите внимание, что, если вы используете `#include` для включения множества исходных файлов в главный файл и имеете в менеджере проекта только главный файл, то в действительности вы имеете только один файл в вашем проекте и не получите преимуществ, описанных выше.

Вы должны поместить прототипы глобальных функций в общие глобальные заголовочные файлы, которые затем включаются другими файлами. Локальные функции должны быть объявлены с ключевым словом `static`, а прототипы этих функций должны быть объявлены или в частном заголовочном файле или в начале исходного файла, в котором они определены. Общие Заголовочные файлы должны также содержать объявления всех глобальных переменных.

Помните, что глобальная переменная может быть **объявлена** во множестве мест, но должна быть **определена** только в одном месте. Один из приемов состоит в размещении условных объявлений в заголовочных файлах. Например, имеется файл `header.h`:

```
#ifndef EXTERN
#define EXTERN extern
#endif

EXTERN int clock_ticks;
```

Затем в одном и только в одном из исходных файлов (скажем `main.c`), вы пишете:

```
#define EXTERN
#include "header.h"
```

Во всех других исходных файлах, пишется только `#include "header.h"` без предшествующего определения `#define`. Так как `main.c` содержит `EXTERN`, не определенный ничем, то включение здесь `header.h` имеет эффект определения глобальной переменной `clock_ticks`. Во всех других исходных файлах, `EXTERN` расширяется как `extern` и таким образом объявляет (но не определяет) `clock_ticks` как внешнюю глобальную переменную, позволяя на нее ссылаться.

6.1.4. Глобальные и локальные переменные, параметры

Функции могут общаться, используя глобальные переменные или параметры функции. В одних процессорах лучше использовать глобальные переменные, в других – локальные переменные и параметры, в третьих процессорах это вообще безразлично. Наши обсуждения целевых процессоров компилятора ImageCraft должны использоваться только как рекомендации. Вы всегда должны сами балансировать необходимость оптимизации с потребностями сопровождения программы.

В общем случае, использование локальных переменных – лучший выбор для Atmel AVR, TI MSP 430 и ARM. Компиляторы ImageCraft для этих процессоров автоматически распределяют локальные переменные в машинных регистрах, если возможно, и в этом случае программы на этих RISC процессорах выполняются намного быстрее. В Motorola HC11 и HC12/ S12, использование локальных переменных дает маленький выигрыш. В HC08/S08, это, вероятно, не имеет значения вообще.

В некоторых процессорах, которые мы не поддерживаем, намного лучше использовать глобальные переменные. Например, Intel 8051 имеет именно такую архитектуру.

6.2. Объявление

Все элементы исходного файла Си должны быть или объявлениями или операторами. Все переменные и имена типов должны быть объявлены прежде, чем они могут быть использованы. Простые объявления данных легко читать и записывать:

```
[<storage class>] typename name;
```

Класс хранения – `storage class`, является опциональным. Это может быть `auto`, `extern` или `register`. Не все имена класса хранения могут появляться в любых объявлениях. Имя типа иногда представляет простой тип:

- `int, unsigned int, unsigned, signed int`
- `short, unsigned short, signed short`
- `char, unsigned char, signed char`
- `float, double` и добавленное в C99 `long double`
- `typedef` – имя производного типа
- `struct <tag>` или `union <tag>`

Для сложных объявлений имеются три дополнительных модификатора типа: массив элементов типа `([])`, функция, возвращающая значение типа `(( ))`, указатель на тип `(*)` и их комбинации, способные сделать объявления трудными для написания и чтения.

6.2.1. Чтение объявлений

Для чтения сложных объявлений используйте правило “вправо-влево”. Начинайте с имени и читайте направо, пока можете, затем двигайтесь влево, пока можете, и затем снова перемещайтесь вправо. Следующий пример демонстрирует этот способ:

```
const int *(*f[5])(int *, char []);
```

Используя правило вправо-влево, вы получаете:

- определив `f` и двигаясь вправо: `f` – массив из 5 ...
- двигаясь влево, `f` – массив из 5 указателей ...
- двигаясь вправо, `f` – массив из 5 указателей на функцию ...
- двигаясь вправо, `f` – массив из 5 указателей на функцию с двумя параметрами (можно пропустить параметры и читать прототип функции позже)...
- двигаясь влево, `f` – массив из 5 указателей на функцию с двумя параметрами, которая возвращает указатель на ...
- двигаясь влево, `f` – массив из 5 указателей на функцию с двумя параметрами, которая возвращает указатель на `int` ...
- двигаясь влево в последний раз, `f` – массив из 5 указателей на функцию с двумя параметрами, которая возвращает указатель на `const int`

Это правило можно также использовать, чтобы писать сложные объявления. В примере используется квалификатор типа `const`. Имеются два типа квалификаторов: `const` (объект доступен только для чтения) и `volatile` (объект может изменяться неожиданным образом).

Квалификатор `volatile` используется для объекта, который может быть изменен асинхронным процессом. Например, глобальная переменная, которая модифицируется процедурой обработки прерывания. Пометка таких переменных как `volatile` указывает компилятору не кэшировать эти значения.

6.2.2. Атомарность доступа

Для большинства 8-разрядных и некоторых из 16-разрядных микроконтроллеров, обращение к 16-разрядному объекту требует двукратного доступа к памяти. Доступ к 32-разрядному длинному объекту требовал бы 4-хкратного доступа, и т.д. По соображениям эффективности, компилятор не отключает прерывания при выполнении многократного доступа. Большую часть времени это работает прекрасно, однако может вызвать проблемы, если вы пишете нечто вроде этого:

```
long var;
void somefunc() { ... if (var != 0) ... }
...
void ISR() { ... if (X) var = 0; else var++; ...}
```

В этом примере `somefunc()` проверяет значение 32-х разрядной переменной, которая модифицируется обработчиком прерывания ISR. В зависимости от того, когда ISR выполняется, возможно, что `somefunc` никогда не обнаружит факт `var == 0`, потому что часть переменной может изменяться во время ее проверки.

Для обхода этой проблемы, вы должны либо не использовать многобайтную переменную этим способом, либо явно запрещать и разрешать прерывания вокруг доступа к переменной, чтобы гарантировать атомарность доступа.

6.2.3. Указатели и массивы

Семантика Си такова, что тип “массив объектов” заменяется указателем на начальный элемент массива объектов этого типа. Это приводит некоторых людей к ошибочному мнению, что указатели и массивы – одно и то же. Их типы часто совместимы, но они – не одно и то же. Например, массив занимает выделенную ему область памяти, в то время как указатель необходимо инициализировать адресом некоторой допустимой области памяти перед доступом к ней.

6.2.4. Типы структура и объединение

По некоторым причинам, некоторые начинающие испытывают затруднения с объявлением `struct`. Основная форма объявления структуры следующая:

```
struct [tag] { member-declaration * } [variable list];
```

Следующие примеры – допустимые формы объявления структурной переменной:

1. `struct { int junk; } var1;`
2. `struct tag1 { int junk; } var2;`
3. ```
   struct tag2;
   struct tag2 { int junk; };
   struct tag2 var3;
   ```

Тег структуры опционален и полезен, если вы хотите повторно сослаться на тот же самый тип `struct` (например, вы можете использовать `struct tag1`, чтобы объявить большее количество переменных этого типа). В Си, даже если два объявления структур в одном и том же файле выглядят идентично, они имеют разные типы `struct`. В примерах выше, все `struct` имеют различные типы, несмотря на то, что выглядят идентично.

Однако, в случае раздельных файлов, это правило ослаблено: если два `struct` имеют то же самое объявление, то они эквивалентны. Это имеет смысл, так как в Си невозможно иметь одно объявление, появляющееся более чем в одном файле. Объявление `struct` в заголовочном файле по-прежнему означает, что в каждом файле, включающем данный заголовочный файл, появляются отдельные (но идентично выглядящие) объявления.
   ```

6.2.5. Прототип функции

В раннем Си, иногда было приемлемо вызывать функцию без предварительного объявления – все будет работать правильно в любом случае. Однако, в компиляторе ImageCraft, важно объявить функцию перед ссылкой на нее, включая типы параметров функции. Иначе возможно, что компилятор будет генерировать неправильный код. Когда вы объявляете функцию с полной информацией о типах аргументов и возвращаемого значения, это называется прототипом функции.

6.3. Выражения и повышение типа

6.3.1. Завершение точкой с запятой

Оператор выражение – один из немногих операторов Си, который требует завершения точкой с запятой. Другие такие операторы – `break`, `continue`, `return`, `goto`, и `do`. Иногда можно увидеть нечто следующее:

```
#define foo blah blah;
...
void foo() { ... };
```

Точка с запятой в конце макроопределения вероятно лишняя и даже может вызвать трудноуловимую ошибку (компиляции или исполнения).

6.3.2. Левое и правое значения

Каждое выражение производит значение. Если выражение находится слева от присваивания, это называется – lvalue – именующим выражением. Во всех других случаях, выражение производит значение переменной – rvalue. Именующее выражение является или именем переменной, или ссылкой на элемент массива, разыменовывает указатель, или элемент поля структуры или объединения; все остальное не является допустимым именующим выражением. Общий вопрос в том, почему компилятор жалуется относительно следующего:

```
((char *)pc)++
```

Ответ – приведение типа не производит именующее выражение. Некоторые компиляторы могут принимать это как расширение, но это – не элемент стандартного Си. Вот пример правильного метода приращения переменной с приведением типа:

```
unsigned pc;
...
pc = (unsigned)((char *)pc + 1);
```

6.3.3. Целые константы

Целочисленные константы могут быть десятичными (по умолчанию), восьмеричными (начинающимися с `0`) или шестнадцатеричными (`0x` или `0X`). Наши компиляторы поддерживают расширение, используя `0b` как префикс двоичных констант. Вы можете явно изменять тип целочисленной константы, добавляя суффиксы `U/u`, `L/l`, или их комбинацию. Тип целого – первый тип каждого списка в следующей таблице, который может содержать значение константы:

Суффикс	Десятичная константа	Восьмеричная/шестнадцатеричная константа
нет	int long int	int unsigned int long int unsigned long int
u или U	unsigned int unsigned long int	unsigned int unsigned long int
l или L	long int	long int unsigned long int
u/U и l/L	unsigned long int	unsigned long int

6.3.4. Выражения

Каждое выражение производит значение и может содержать побочные эффекты. В стандартном Си вы можете смешивать и согласовывать выражения различных типов, и по некоторым правилам компилятор преобразует выражения в правильный тип. Целое выражение и выражение с плавающей точкой могут использоваться вместе, и в большинстве случаев будет достигнут ожидаемый результат. Неожиданный результат может получиться там, где тип выражения зависит исключительно от типов операндов, а не от способа их использования. Например:

```
long_var = int_var1 * int_var2; // int multiply
long_var = (long)int_var1 * int_var2; // long multiply
```

Первое умножение выполняется как умножение целых чисел, а не длинных. Если вы хотите произвести умножение длинных чисел, по крайней мере, один из операндов должен иметь тип `long`, как видно из второго примера. Это применяется также к присваиванию значений переменным с плавающей точкой и другим.

Другое замечание состоит в том, что по стандарту Си, операнды повышаются до эквивалентных типов, прежде чем операция будет выполнена. В частности целочисленное выражение должно быть повышено, по крайней мере, до типа `int`, если его тип меньше чем тип `int`. Однако повышение не обязано происходить физически, если дает тот же самый результат. Наши компиляторы пробуют оптимизировать байтовые операции везде, где возможно. Некоторые выражения более трудны для оптимизации, особенно если они производят промежуточное значение. Например, компилятор не может оптимизировать следующее, так как `*p` – временное значение, которое должно быть сохранено:

```
char *p;
...
... *p++...
```

6.3.5. Операции

Си имеет богатый набор операторов, включая поразрядные, упрощающие обработку регистров ввода/вывода (см. 8.2. Манипуляция битами). В языке не имеется “логического” или “булева” типа, так что любое ненулевое значение принимается как “истина”. Вы можете смешивать в выражении любые операторы, включая логический, поразрядный и т.д. Следующая таблица содержит список операторов в порядке убывания приоритета. В пределах каждого ряда операторы имеют одинаковый приоритет.

<table>
<tr><th>Символ</th><th>Оператор</th><th>Ассоциативность</th></tr>
<tr><td>( )
[ ]
->
.</td><td>вызов функции
элемент массива
разыменование указателя на поле структуры
ссылка на поле структуры</td><td>слева направо</td></tr>
<tr><td>!
~
++
--
+
-
*
&
(type)
sizeof</td><td>логическое не
дополнение до единицы
пред/пост инкремент
пред/пост декремент
унарный плюс
унарный минус
разыменование указателя
взятие адреса
приведение типа
размер типа</td><td>справа налево</td></tr>
<tr><td>*
/
%</td><td>умножение
деление
остаток</td><td>слева направо</td></tr>
<tr><td>+
-</td><td>сложение
вычитание</td><td>слева направо</td></tr>
<tr><td><<
>></td><td>левый сдвиг
правый сдвиг [a)]</td><td>слева направо</td></tr>
<tr><td><
<=
>
>=</td><td>меньше чем
меньше чем или равно
больше чем
больше чем или равно</td><td>слева направо</td></tr>
<tr><td>==
!=</td><td>Равно
не равно</td><td>слева направо</td></tr>
<tr><td>&</td><td>поразрядное и</td><td>слева направо</td></tr>
<tr><td>^</td><td>поразрядное исключающее или</td><td>слева направо</td></tr>
<tr><td>|</td><td>поразрядное или</td><td>слева направо</td></tr>
<tr><td>&&</td><td>логическое и</td><td>слева направо</td></tr>
<tr><td>||</td><td>логическое или</td><td>слева направо</td></tr>
<tr><td>?:</td><td>условное выражение с 3-мя операндами</td><td>справа налево</td></tr>
<tr><td>= += -= *= /=
%= &= ^= |=
<<= >>=</td><td>операторы присваивания</td><td>справа налево</td></tr>
<tr><td>,</td><td>оператор запятая</td><td>слева направо</td></tr>
</table>

a). Стандартный Си не определяет, является ли правый сдвиг арифметическим или логическим. Все компиляторы ImageCraft используют арифметический сдвиг для знакового операнда и логический для беззнакового операнда.

Злоупотребление макроопределениями

Некоторые люди используют #define, чтобы назначать “лучшие имена” для некоторых операторов. Например, EQ вместо ==, BITAND вместо &, и т.д. Такая практика – вообще плохая идея, так как служит только для создания персонального диалекта языка, делая программу более трудной в поддержке и чтении другими людьми.

Опасные операторы

- Ошибочное использование оператора = вместо оператора ==. Напоминает порочную практику злоупотребления макроопределениями. Пишите тщательней или используйте инструментарий способный отлавливать подобные ошибки.
- Поразрядные операторы имеют более высокий приоритет, чем логические операторы. Для многих программистов, Си представляет идеальную смесь конструкций высокого уровня с возможностью доступа к низкому уровню. Однако есть один случай, где даже изобретатели Си признают, что это – ошибочная особенность. Это означает, что вы должны писать:

  ```
  if ((flags & bit1) != 0 && ...
  ```

  с “дополнительным” набором круглых скобок, чтобы получить правильную семантику. К сожалению, сила требований обратной совместимости такова, что даже C++ должен сохранять эту ошибку.

6.4. Операторы

Следующие словосочетания `if-body`, `while-body`, `for-body` ... и т.д. означают тело соответствующих операторов Си.

6.4.1. Оператор выражение

`[ label: ] [expression];`

См. 6.3. Выражения и повышение типа для обсуждения выражений. Пустая одиночная точка с запятой является оператором нуль-выражения.

6.4.2. Составной оператор

`{ [statement]* }`

Составной оператор – последовательность из нуля или большего количества операторов, заключенных в пару фигурных скобок { }. Локальные объявления допустимы только немедленно после открывающей скобки и до любого выполнимого оператора, и иногда скобки вводятся только для цели объявления временных локальных переменных.

6.4.3. Оператор If

`if (<expr>) if-body [ else else-body ]`

Если результат вычисления `<expr>` ненулевой, то выполняется `if-body`. Иначе, выполняется `else-body`, если существует. Отсутствует проблема "повисшего `else`", поскольку ключевое слово `else` всегда ассоциируется с ближайшим предшествующим ключевым словом `if`.

6.4.4. Оператор While

`while (<expr>) while-body`

Тело `while-body` выполняется, пока результат вычисления `<expr>` ненулевой. Обратите внимание, что наши компиляторы транслируют это в подобное следующему:

```
goto bottom
loop_top: <while-body>
bottom: if <expr> goto loop_top
```

Это не столь очевидно, но по сравнению с помещением проверки в верхней части, эта последовательность выполняет `n + 2` ветвлений для цикла, который выполняется `n` раз, против `2n + 1` ветвлений для более очевидного размещения проверки.

6.4.5. Оператор For

`for ( [<expr1>] ; <expr>; <expr2> ) for-body`

Тело `for-body` выполняется пока результат вычисления `<expr>` ненулевой. `<expr2>` выполняется после `for-body`. `<expr1>` и `<expr2>` – места, где вы обычно поместили бы начальные выражения и приращения цикла соответственно.

6.4.6. Оператор Do

`do do-body while (<expr>);`

Выполняет `do-body`, по меньшей мере, один раз и если вычисление выражения `<expr>` дает ненулевой результат, то процесс повторяется.

6.4.7. Оператор Break

```
break;
```

Допустимо только внутри тела цикла или внутри оператора `switch`. Заставляет передавать управление за пределы цикла или переключателя. Внутри переключателя, выполнение проваливается к следующему `case`, если не завершается оператором `break`.

6.4.8. Оператор Continue

```
continue;
```

Допустимо только внутри тела цикла. Это заставляет передавать управление проверке цикла. Внутри оператора `for` будет пропущено обычно выполняемое третье выражение.

6.4.9. Оператор Goto

```
goto label;
```

Передает управление метке `label`. Не имеется никаких ограничений на то, где размещена метка, пока это – допустимая метка внутри той же самой функции. Это обычно не является хорошей идеей, т.к. оператор позволяет перейти в середину цикла или в другие “плохие” места.

6.4.10. Оператор Return

```
return [<expr>];
```

Возвращает управление обратно в вызывающую функцию и опционально возвращает значение определенного выражения.

6.4.11. Оператор Switch

```
switch (<int expr>) switch-body
```

Вычисляет целочисленное выражение и передает управление метке выбора внутри `switch-body`, имеющей то же значение, что и выражение. Если соответствия не имеется и имеется заданная по умолчанию метка, то управление передается метке выбора по умолчанию. Обратите внимание, что хотя `switch-body` обычно пишется так:

```
{ case <int>: [expression;] * ... default: [expression;] * }
```

язык Си не навязывает этот формат. Метка выбора и заданная по умолчанию метка могут появляться только внутри `switch-body`. Другая важная особенность – выполнение проваливается к следующей метке выбора, если не завершается оператором `break`.

7. БИБЛИОТЕКА Си И ФАЙЛ ЗАПУСКА

7.1. Замена библиотечной функции

Вы можете написать вашу собственную версию библиотечной функции. Например, вы можете реализовать вашу собственную функцию `putchar()`, чтобы сделать вывод на устройство LCD. Исходный код библиотеки доступен, и его можно использовать как образец. Заменить заданную по умолчанию библиотечную функцию можно одним из следующих методов:

- Вы можете включить вашу функцию в один из ваших файлов проекта. В этом случае система компилятора не будет использовать библиотечную функцию. Обратите внимание, что в этом случае, в отличие от библиотечного модуля, ваша функция всегда будет включаться в выходной файл программы, даже если вы ее не используете.
- Вы можете создать вашу собственную библиотеку. Подробнее см. 12.8. Библиотекарь.
- Вы можете заменить заданную по умолчанию версию библиотеки вашей собственной. Обратите внимание, что при обновлении компилятора до новой версии, вы должны сделать эту замену снова. См. 12.8. Библиотекарь о замене библиотечного модуля.

7.2. Файл запуска

Компоновщик вставляет код файла запуска перед кодом ваших файлов и связывает стандартную библиотеку libc430.a с вашей программой. Файл запуска имеет имя crt430.o. Файл запуска определяет глобальный символ _cstart, являющийся стартовой точкой вашей программы. Функции файла запуска:

1. Инициализировать указатель стека.
2. Копировать инициализированные данные из области idata в область данных.
3. Инициализировать область bss нулевыми значениями.
4. Вызвать процедуру пользователя main.
5. Определить точку входа exit как бесконечный цикл. Если из main когда-либо произойдет возврат, система останется в этом месте в бесконечном цикле.

Файл запуска также определяет вектор сброса. Чтобы использовать другие прерывания изменять файл запуска не требуется. Компиляция или ассемблирование файла запуска требует специального ключа ассемблера (-n). Вы можете использовать среду для компиляции файла запуска, используя команду *File>Compile File...>Startup File To Object*.

Модификация и использование нового файла запуска:

```
cd \iccv7430\libsrc.430 ; or wherever you install the compiler
<edit crt430.s>
<open crt430.s using IDE>
<choose Compile File>To Object>; generate new crt430.o>
copy crt430.o ..\lib ; copy to the library directory
```

Вы также можете иметь несколько файлов запуска. Определите имя файла запуска в диалоговом окне опций проекта. Заметьте, что файл запуска должен быть определен с абсолютным путем либо должен находиться в директории, определенной в путях к библиотекам в опциях проекта.

7.3. Общее описание библиотеки Си

7.3.1. Исходный код библиотеки

Исходный текст библиотеки (`c:\iccv7430\libsrc.430\libsrc.zip` по умолчанию) – это защищенный паролем zip-файл. Для разархивации необходима программа unzip, доступная во многих местах сети, если вы еще не имеете такой. Пароль можно найти в зарегистрированной версии в меню *About*. Пример разархивации библиотеки:

```
cd \iccv7430\libsrc
unzip -s libsrc.zip
; unzip prompts for password
```

7.3.2. Функции, специфичные для MSP430

Будущие версии ICC430 будут включать функции, специфичные для подсистем, т.е. для доступа к UART, SPI и т.д. Однако для генерации кода доступа к периферии может быть использован Application Builder.

`msp*.h (msp430x, ... etc.)`

Файл содержит определения регистров ввода/вывода и смещений векторов прерываний.

`msp430def.h`

Файл содержит полезные макросы и определения.

7.3.3. Другие заголовочные файлы

Поддерживаются следующие ниже стандартные заголовочные файлы Си. Вообще, хорошим стилем является включать заголовочные файлы, если вы используете перечисленные функции в вашей программе. В случае с плавающей точкой и длинными целыми, вы обязаны включать заголовочные файлы, так как компилятор должен знать их прототипы. См. 9.2. Интерфейс ассемблера и соглашения о вызовах.

`assert.h` – макрос диагностики.

`ctype.h` – функции символьного типа.

`float.h` – характеристики формата плавающей точки.

`limits.h` – размеры и диапазоны типов данных.

`math.h` – математические функции с плавающей точкой.

`stdarg.h` – поддержка функций с переменными параметрами.

`stddef.h` – стандартные определения.

`stdio.h` – стандартные функции Ввода/Вывода.

`stdlib.h` – стандартная библиотека, включая функции распределения памяти.

`string.h` – функции манипулирования строками.

7.4. Функции символьного типа

Следующие функции категоризируют ввод согласно набору символов ASCII. Включите в исходный файл `#include <ctype.h>` перед использованием этих функций.

- `int` **`isalnum`**`(int c)`

  Возвращает не нуль, если `c` – цифра или алфавитный символ.

- `int` **`isalpha`**`(int c)`

  Возвращает не нуль, если `c` – алфавитный символ.

- `int` **`iscntrl`**`(int c)`

  Возвращает не нуль, если `c` – управляющий символ (например, `FF`, `BELL`, `LF`).

- `int` **`isdigit`**`(int c)`

  Возвращает не нуль, если `c` – цифра.

- `int` **`isgraph`**`(int c)`

  Возвращает не нуль, если `c` – печатный символ и не пробел.

- `int` **`islower`**`(int c)`

  Возвращает не нуль, если `c` – алфавитный символ нижнего регистра.

- `int` **`isprint`**`(int c)`

  Возвращает не нуль, если `c` – печатный символ.

- `int` **`ispunct`**`(int c)`

  Возвращает не нуль, если `c` – печатаемый символ и не пробел, не цифра, не алфавитный символ.

- `int` **`isspace`**`(int c)`

  Возвращает не нуль, если `c` – пробельный символ, включая пробел, `CR`, `FF`, `HT`, `NL`, и `VT`.

- `int` **`isupper`**`(int c)`

  Возвращает не нуль, если `c` – алфавитный символ верхнего регистра.

- `int` **`isxdigit`**`(int c)`

  Возвращает не нуль, если `c` – шестнадцатеричная цифра.

- `int` **`tolower`**`(int c)`

  Возвращает `c` в нижнем регистре, если `c` – символ верхнего регистра. Иначе возвращает `c`.

- `int` **`toupper`**`(int c)`

  Возвращает `c` в верхнем регистре, если `c` – символ нижнего регистра. Иначе возвращает `c`.

7.5. Математические функции с плавающей точкой

Поддерживаются следующие математические процедуры с плавающей точкой. Вы должны включить в исходный файл `#include <math.h>` перед использованием этих функций.

- `float asinf(float x)`

  Возвращает арксинус `x` для `x` в радианах.

- `float acosf(float x)`

  Возвращает арккосинус `x` для `x` в радианах.

- `float atanf(float x)`

  Возвращает арктангенс `x` для `x` в радианах.

- `float atan2f(float x, float y)`

  Возвращает угол в диапазоне `[-pi, +pi]` радиан, чей тангенс равен `y/x`.

- `float ceilf(float x)`

  Возвращает наименьшее целое число не меньшее чем `x`.

- `float cosf(float x)`

  Возвращает косинус `x` для `x` в радианах.

- `float coshf(float x)`

  Возвращает гиперболический косинус `x` для `x` в радианах..

- `float expf(float x)`

  Возвращает `e` в степени `x`.

- `float exp10f(float x)`

  Возвращает `10` в степени `x`.

- `float fabsf(float x)`

  Возвращает абсолютное значение `x`.

- `float floorf(float x)`

  Возвращает наибольшее целое число не большее чем `x`.

- `float fmodf(float x, float y)`

  Возвращает остаток от `x/y`.

- `float frexpf(float x, int *pexp)`

  Возвращает дробь `f` и сохраняет целое – степень числа `2` в `*pexp`, представляющие входное значение `x`. Возвращаемое значение находится в интервале `[1/2, 1)`. Величина `x=f*2**(*pexp)`.

- `float froundf(float x)`

  Округляет `x` до самого близкого целого числа.

- `float ldexpf(float x, int exp)`

  Возвращает `x*2**exp`.

- `float logf(float x)`

  Возвращает натуральный логарифм `x`.

- `float` **`log10f`**`(float x)`

  Возвращает логарифм `x` по основанию `10`.

- `float` **`modff`**`(float x, float *pint)`

  Возвращает дробь `f` и сохраняет целое число в `*pint`. Значение `x=f+(*pint)`. Величина `abs(f)` находится в интервале `[0, 1)`, оба `f` и `*pint` имеют знак `x`.

- `float` **`powf`**`(float x, float y)`

  Возвращает `x` в степени `y`.

- `float` **`sqrtf`**`(float x)`

  Возвращает квадратный корень `x`.

- `float` **`sinf`**`(float x)`

  Возвращает синус `x` для `x` в радианах.

- `float` **`sinhf`**`(float x)`

  Возвращает гиперболический синус `x` для `x` в радианах.

- `float` **`tanf`**`(float x)`

  Возвращает тангенс `x` для `x` в радианах.

- `float` **`tanhf`**`(float x)`

  Возвращает гиперболический тангенс `x` для `x` в радианах.

Так как типы `float` и `double` имеют тот же самый размер (32 бита), `math.h` также содержит набор макросов, которые отображают имена функций на имена без суффикса `f`, например `pow` – то же, что и `powf`, `sin` – то же, что и `sinf` и т.д.

7.6. Стандартные функции ввода/вывода

Так как стандартный файл ввода/вывода для встроенного микроконтроллера не имеет смысла, многое из содержания стандартного `stdio.h` неприменимо. Однако некоторые функции ввода/вывода поддерживаются. Используйте `#include <stdio.h>` перед использованием этих функций. Вы сами должны инициализировать порты ввода/вывода. Самый нижний уровень процедур ввода/вывода состоит из процедур символьного ввода (`getchar`) и вывода (`putchar`). Вы будете должны реализовать функцию `putchar`, специфичную для вашего устройства (и `getchar`, если вы используете функции ввода STDIO). Имеются некоторые типовые процедуры в `c:\iccv7430\examples.430\` для устройств UART. Вы можете начать с одного из примеров и добавить его в ваш список файлов проекта.

7.6.1. Вывод возврата каретки

По умолчанию, функция посимвольного вывода `putchar` посылает символ устройству UART без модификации. Однако, при выводе, чтобы появиться, как ожидается в программе терминала Windows, символ `'\n'` должен быть преобразован в пару символов возврата каретки и перевода строки (`CR/LF`). Это может быть выполнено, используя следующее:

```
extern int _textmode; // this is defined in the library
...
_textmode = 1;
```

Если это присваивание выполнено, то `putchar` отобразит символ `'\n'` в пару `CR/LF`. Вы можете отменить это поведение, присвоив указанной переменной нулевое значение.

7.6.2. Использование Printf с несколькими устройствами

Использовать `printf` с несколькими устройствами очень просто. Вы можете написать вашу собственную функцию `putchar()` для вывода на различные устройства в зависимости от глобальной переменной и функции, которая изменяет эту переменную. Переключение ввода/вывода между различными устройствами может быть обеспечено специальной функцией перенаправления. Вы можете даже реализовать версию `printf`, которая принимает некоторый параметр номера устройства, используя функцию `vfprintf()`, описанную ниже.

7.6.3. Список стандартных функций ввода/вывода

- `int` **`getchar`**`()`

  Возвращает символ из UART, используя режим опроса.

- `int` **`printf`**`(char *fmt, ..)`

  Выводит форматированный текст согласно спецификаторам формата в строке формата `fmt`. **ЗАМЕЧАНИЕ: `printf` поддерживается в трех версиях**, в зависимости от размера вашего кода и особых требований (большее количество возможностей – больше размер кода):

  - Базовый, только следующие спецификаторы формата без модификаторов: `%c`, `%d`, `%x`, `%u` и `%s`.
  - Длинный, длинные модификаторы в дополнение к полям точности и ширины: `%ld`, `%lu`, `%lx`.
  - Плавающий: все форматы, включая `%f` для плавающей точки.

  Размер кода значительно увеличивается при продвижении вниз по списку. Выбирайте версию для использования в 4.13. Опции компилятора: Целевое устройство.

  Спецификаторы формата являются подмножеством стандартных форматов:

  `%[flags]*[width][.precision][l]<conversion character>`

Флаги формата:

`#` – альтернативная форма. Для преобразования `x` или `X`, генерируются `0x` или `0X`. Для преобразования чисел с плавающей точкой генерируется десятичная точка, даже если число с плавающей точкой может быть преобразовано точно в целое число.

`-` (минус) – выравнивание по левому краю поля вывода.

`+` (плюс) – добавляет символ '+' для положительного целого числа.

`' '` (пробел) – использует пробел как символ знака для положительного целого числа.

`0` – заполнять нулями вместо пробелов.

Ширина задается или десятичным целым или `'*'`, означая, что значение берется из следующего параметра. Ширина определяет минимальное число символов для вывода, выравнивание влево или вправо, если необходимо, и заполнено пробелами или нулями, в зависимости от символов флагов.

Точность предваряется `'.'` и является или десятичным целым или `'*'`, означая, что значение принимается из следующего параметра. Точность определяет минимальное число цифр для целочисленного преобразования, максимальное число символов для `'s'` – строкового преобразования и число цифр после десятичной точки для преобразования с плавающей точкой.

Символы преобразования следующие. Если `l` (буква эль) появляется перед символом целочисленного преобразования, то параметр принимается как длинное целое число.

`d` – печатать следующий параметр как десятичное целое число

`o` – печатать следующий параметр как восьмеричное целое число без знака

`x` – печатать следующий параметр как шестнадцатеричное целое число без знака

`X` – также как x за исключением того, что для 'A'-'F' используется верхний регистр

`u` – печатать следующий параметр как десятичное целое число без знака

`s` – печатать следующий параметр как Си-строку с нуль-терминатором

`c` – печатать следующий параметр как символ ASCII

`f` – печатать следующий параметр как десятичное число с плавающей точкой (например 31415.9)

`e` – печатать следующий параметр как число с плавающей точкой в экспоненциальном формате (например 3.14159e4)

`g` – печатать следующий параметр как число с плавающей точкой, или в десятичном или в экспоненциальном формате, какой более удобен.

- `int` **`putchar`**`(int c)`

  Печатать одиночный символ. Процедура библиотеки использует UART в режиме опроса, для вывода одиночного символа. См. “Примечание” выше относительно вывода символа '\n' в программу терминала Windows. Переопределите эту функцию, если хотите направить вывод (из printf и т.д.) на устройство по вашему выбору.

- `int` **`puts`**`(char *s)`

  Печатать строку, сопровождаемую NL.

- `int` **`sprintf`**`(char *buf, char *fmt)`

  Печатает форматированный текст в `buf` согласно спецификаторам формата в `fmt`. Спецификаторы формата – такие же, как в `printf()`.

- `int scanf(char *fmt, ...)`

  Читает ввод согласно формату строки `fmt`. Чтобы читать ввод используется функция `getchar()`. Следовательно, если вы переопределяете функцию `getchar()`, вы можете использовать эту функцию, чтобы читать из любого устройства, которое вы выбираете.

  Непробельные пробельные символы в строке формата должны точно соответствовать входным и пробельным символам согласно самой длинной последовательности (включая нулевой размер строки) пробельных символов ввода. Символ `%` представляет спецификатор формата:

  - `[l]` – длинный модификатор. Этот опциональный модификатор определяет, что соответствующий параметр имеет тип указатель на длинное целое
  - `d` – ввод – десятичное число. Параметр должен быть указателем на (`long`) `int`.
  - `x/X` – ввод – шестнадцатеричное число, возможно начинающееся с `0x` или `0X`. Параметр должен быть указателем на (`long`) `int` без знака.
  - `u` – ввод – десятичное число. Параметр должен быть указатель на (`long`) `int` без знака.
  - `o` – ввод – десятичное число. Параметр должен быть указатель на (`long`) `int` без знака.
  - `c` – ввод – символ. Параметр должен быть указателем на символ.

- `int sscanf(char *buf, char *fmt, ...)`

  То же самое, что `scanf` за исключением того, что ввод принимается из буфера `buf`.

- `int vprintf(char *fmt, va_list va)`

  то же, что `printf` за исключением того, что параметры после строки формата специфицируются, используя механизм `stdarg`.

7.7. Стандартная библиотека и функции памяти

Заголовочный файл стандартной библиотеки <stdlib.h> определяет макросы NULL, RAND_MAX, typedefs size_t и объявляет следующие ниже функции. Обратите внимание, что вы должны инициализировать кучу вызовом _NewHeap перед использованием любой из процедур распределения памяти (calloc, malloc, и realloc).

- `int abs(int i)`

  Возвращает абсолютное значение i.

- `int atoi(char *s)`

  Преобразовывает строку s в целое число, или возвращает 0, если происходит ошибка.

- `double atof(const char *s)`

  Преобразовывает строку s в double и возвращает его.

- `long atol(char *s)`

  Преобразовывает строку s в long, или возвращает 0, если происходит ошибка.

- `void *calloc(size_t nelem, size_t size)`

  Выделяет фрагмент памяти, достаточно большой, чтобы вместить nelem объектов, каждый из которых размера size. Память инициализируется нулями. Возвращает 0, если не может удовлетворить запрос.

- `void exit(status)`

  Завершает программу. В среде встроенного контроллера, это обычно просто бесконечный цикл и в основном используется как точка возврата из пользовательской функции main.

- `void free(void *ptr)`

  Освобождает предварительно выделенную память кучи.

- `char *ftoa(float f, int *status)`

  Преобразовывает число с плавающей точкой в его представление в ASCII. Возвращает статический буфер приблизительно из 15 символов. Если число находится вне диапазона, *status устанавливается в константу _FTOA_TOO_LARGE или _FTOA_TOO_SMALL, определенные в stdlib.h, и возвращается 0. Иначе, *status устанавливается в 0, и возвращается буфер char. Это быстрая версия ftoa, но она не может обрабатывать значения вне перечисленного диапазона. Пожалуйста, свяжитесь с нами, если нуждаетесь в версии, обрабатывающей более широкой диапазон.

  Как в большинстве других функций Си с подобным прототипом, *status предполагает, что вы должны передать этой функции адрес переменной. Не объявляйте переменную указатель, и передавайте переменную без инициализации значения указателя.

- `void itoa(char *buf, int value, int base)`

  Преобразовывает значение целого числа со знаком в строку ASCII, используя base как основание системы счисления. Основание может быть целым числом от 2 до 36.

- `void ltoa(char *buf, long value, int base)`

  Преобразовывает длинное значение в строку ASCII, используя base как основание системы счисления.

- `void utoa(char *buf, unsigned value, int base)`

  То же что itoa за исключением того, что параметр принимается как int без знака.

- `void **ultoa**(char *buf, unsigned long value, int base)`

  То же что `ltoa` за исключением того, что параметр принимается как длинное без знака.

- `void ***malloc**(size_t size)`

  Выделяет фрагмент памяти размера `size` из кучи. Возвращает 0, если не может удовлетворить запрос.

- `void **_NewHeap**(void *start, void *end)`

  Инициализирует кучу для процедур распределения памяти. `malloc` и связанные процедуры управляют памятью в области кучи. См. 9.5. Области программы относительно функций размещения памяти. Типичное обращение использует адрес символа `_bss_end + 1` как значение “`start`”. Символ `_bss_end` определяет конец памяти данных, используемой компилятором для глобальных переменных и строк.

  ```
  extern char _bss_end;
  _NewHeap(&_bss_end+1, &_bss_end + 201); // heap 200 bytes
  ```

  Учтите, что для микроконтроллера с малым объемом памяти данных часто невозможно или слишком сложно использовать динамическую память из-за перерасхода и потенциальной фрагментации памяти. Часто простой статически распределенный массив служит лучшим решением для удовлетворения таких потребностей.

- `int **rand**(void)`

  Возвращает псевдослучайное число между 0 и `RAND_MAX`.

- `void ***realloc**(void *ptr, size_t size)`

  Перераспределяет предварительно выделенный фрагмент памяти с новым размером.

- `void **srand**(unsigned seed)`

  Инициализирует начальное значение случайного числа для последующих вызовов `rand()`.

- `long **strtol**(char *s, char **endptr, int base)`

  Преобразовывает строку `s` в длинное целое число по основанию `base`. Если `base` равно 0, то `strtol` выбирает `base` в зависимости от начальных символов (после опционального знака "минус" если он есть) в строке `s`. `0x` или `0X` указывает на шестнадцатеричное целое число, 0 указывает на восьмеричное целое число, или в противном случае принимается десятичное целое. Если `endptr` – не `NULL`, то `*endptr` будет установлен адресом, где в `s` заканчивается преобразование.

- `unsigned long **strtoul**(char *s, char **endptr, int base)`

  Является аналогом `strtol` за исключением того, что возвращаемый тип – беззнаковое длинное.

7.8. Строковые функции

Поддерживаются следующие строковые функции. Используйте `#include <string.h>` перед использованием этих функций. Файл `<string.h>` определяет `NULL` и `typedefs size_t`, и следующие функции со строками и символьными массивами:

- `void *memchr(void *s, int c, size_t n)`

  Поиск первого местонахождения `c` в массиве `s` размером `n`. Возвращает адрес соответствующего элемента или пустой указатель, если соответствие не найдено.

- `int memcmp(void *s1, void *s2, size_t n)`

  Сравнивает два массива, каждый из которых размером `n`. Возвращает 0, если массивы равны и больше чем 0, если первый отличающийся элемент в `s1` больше чем соответствующий элемент в `s2`. Иначе, возвращает число меньшее, чем 0.

- `void *memcpy(void *s1, void *s2, size_t n)`

  Копирует `n` байт из `s2` в `s1`.

- `void *memmove(void *s1, void *s2, size_t n)`

  Копирует `s2` в `s1`, размером `n` каждый. Процедура работает правильно, даже если массивы накладываются. Возвращает `s1`.

- `void *memset(void *s, int c, size_t n)`

  Сохраняет `c` во всех элементах массива `s` размером `n`. Возвращает `s`.

- `char *strcat(char *s1, char *s2)`

  Конкатенирует `s2` к `s1`. Возвращает `s1`.

- `char *strchr(char *s, int c)`

  Поиск первого местонахождения `c` в `s`, включая нуль-терминатор. Возвращает адрес соответствующего элемента или пустой указатель, если соответствие не найдено.

- `int strcmp(char *s1, char *s2)`

  Сравнивает две строки. Возвращает 0, если строки равны, положительное число, если первый отличный элемент в `s1` больше чем соответствующий элемент в `s2`. Иначе, возвращает отрицательное число.

- `char *strcpy(char *s1, char *s2) )`

  Копирует `s2` в `s1`. Возвращает `s1`.

- `size_t strcspn(char *s1, char *s2)`

  Поиск первого элемента в `s1`, который соответствует любому из элементов в `s2`. Нуль-терминаторы рассматриваются как части строки. Возвращает индекс `s1`, с которым найдено соответствие.

- `size_t strlen(char *s)`

  Возвращает длину `s`. Нуль-терминатор не считается.

- `char *strncat(char *s1, char *s2, size_t n)`

  Конкатенирует до `n` элементов `s2` в `s1`, исключая нуль-терминатор. Затем копирует нуль-терминатор в конец `s1`. Возвращает `s1`.

- `int strncmp(char *s1, char *s2, size_t n)`

  Также как функция `strcmp` за исключением того, что сравнивает не более `n` символов.

- `char *`**`strncpy`**`(char *s1, char *s2, size_t n)`

  Также как функция `strcpy` за исключением того, что копирует не более `n` символов.

- `char *`**`strpbrk`**`(char *s1, char *s2)`

  Делает тот же самый поиск, что и функция `strcspn`, но возвращает указатель на соответствующий элемент в `s1`, если элемент – не нуль-терминатор. Иначе, возвращает пустой указатель.

- `char *`**`strrchr`**`(char *s, int c)`

  Поиск последнего местонахождения `c` в `s` и возврат указателя на него. Возвращает пустой указатель, если соответствие не найдено.

- `size_t `**`strspn`**`(char *s1, char *s2)`

  Поиск первого элемента в `s1`, который не соответствует никакому из элементов в `s2`. Нуль-терминатор `s2` рассматривается как часть `s2`. Возвращает индекс, где условие выполняется.

- `char *`**`strstr`**`(char *s1, char *s2)`

  Находит в `s1` подстроку, которой соответствует `s2`. Возвращает адрес подстроки в `s1` если соответствие найдено или иначе пустой указатель.

7.9. Функции с переменными параметрами

Файл `<stdarg.h>` обеспечивает поддержку обработки функций с параметрами, число и тип которых заранее не известны. Он определяет псевдо-тип `va_list` и три макроса:

- **`va_start`**`(va_list foo, <last-arg>)`

  Инициализирует переменную `foo`.

- **`va_arg`**`(va_list foo, <promoted type>)`

  Обращается к следующему параметру, приводит к специфицированному типу. Обратите внимание, что тип должен быть “расширенный тип”, типа `int`, `long` или `double`. Меньшие целочисленные типы, типа `char` недопустимы и дадут неправильные результаты.

- **`va_end`**`(va_list foo)`

  Заканчивает обработку переменных параметров.

Например, функция `printf()` может быть реализована, используя `vfprintf()`, следующим образом:

```
#include <stdarg.h>

int printf(char *fmt, ...)
    {
    va_list ap;

    va_start(ap, fmt);
    vfprintf(fmt, ap);
    va_end(ap);
    }
```

8. ПРОГРАММИРОВАНИЕ MSP430

8.1. Доступ к специфическим ресурсам MSP430

Сила Си в том, что, являясь языком высокого уровня, он позволяет вам обращаться к ресурсам низкого уровня целевых устройств. С такими способностями, имеется очень немного причин использовать ассемблер за исключением случаев, где крайне важен максимально оптимизированный код. Даже в случаях, когда низкоуровневые возможности не доступны на Си, обычно встроенный ассемблер и макроопределения препроцессора позволяют получить прозрачный доступ к этим средствам.

Заголовочные файлы `msp430*.h` (такие как `msp430x14x.h`) определяют 8.4. Регистры ввода/вывода MSP430, специфические для типа устройства. Файл `msp430def.h` определяет много полезных макросов.

Примечание: наши заголовочные файлы определяют биты регистров ввода/вывода как битовые маски для совместимости с примерами в документации TI и с заголовочными файлами IAR C. Например:

```
#include <msp430x14x.h>
IE |= WDTIE;
```

8.2. Манипуляция битами

Частая задача при программировании микроконтроллеров состоит в установке или сбросе некоторых битов в регистрах ввода/вывода. К счастью стандартный Си хорошо подходит для операций с битами без использования команд ассемблера или других нестандартных конструкций. Си определяет некоторые полезные поразрядные операторы:

- **a | b** – поразрядное "ИЛИ". Выражения, обозначенные – a и b, поразрядно логически складываются. Это используется, чтобы установить некоторые биты, особенно когда используется в форме присваивания |=. Например:

```
PORTA |= 0x80; // turn on bit 7 (msb)
```

- **a & b** – поразрядное "И". Это полезно для проверки того, что некоторые биты установлены. Например:

```
if ((PINA & 0x81) == 0) // check bit 7 and bit 0
```

  Обратите внимание, что круглые скобки необходимы вокруг выражений оператора &, так как он имеет более низкий приоритет, чем оператор ==. Обратите внимание на использование PINA вместо PORTA, чтобы прочитать порт.

- **a ^ b** – поразрядное "исключающее ИЛИ". Этот оператор полезен для дополнения бита. Например, в следующем случае, бит 7 инвертируется:

```
PORTA ^= 0x80; // flip bit 7
```

- **~a** – поразрядное дополнение. Этот оператор дополняет выражение до 1 – инвертирует биты. Это особенно полезно вместе с поразрядным "И", чтобы сбросить некоторые биты:

```
PORTA &= ~0x80; // turn off bit 7
```

  Для этих операций компилятор генерирует оптимальные машинные команды. Например, некоторые команды могли бы использоваться для поразрядного оператора "И" для условного перехода, основанного на состоянии разряда.

8.2.1. Битовые макросы

Некоторые примеры макросов, которые могут быть полезны для манипуляций с битами:

```
#define SetBit(x,y)     (x|=(1<<y))
#define ClrBit(x,y)     (x&=~(1<<y))
#define ToggleBit(x,y)  (x^=(1<<y))
#define FlipBit(x,y)    (x^=(1<<y)) // Same as ToggleBit.
#define TestBit(x,y)    (x&(1<<y))
```

8.2.2. Манипуляция битами, bit-переменная и битовое поле

Некоторые компиляторы поддерживают расширения Си для доступа к отдельным битам. Например, PORTA.2, для доступа к биту 2 регистра PORTA. По определению, расширения не переносимы в другие стандартные компиляторы Си. Также заметим, что операции манипуляции битами, перечисленные здесь, производят лучший код и полностью переносимы. Предложенные выше макросы могут облегчить их использование. Поэтому, наши компиляторы не поддерживают это расширение.

Некоторые пользователи хотят использовать битовые поля структур для доступа к битам регистра. Хотя это работало бы с указателем на структуру и с подходящим приведением его типа к корректному адресу, это требует расширение языка для оверлея типа структуры со специфическим адресом регистра. Также, порядок битов не определен языком Си, и обычно битовые поля генерируют не лучший код. Мы настоятельно рекомендуем использовать побитовые операторы вместо этого.

8.3. Встроенный ассемблер

Кроме написания ассемблерных функций в ассемблерных файлах, встроенный ассемблер позволяет вам писать ассемблерный код внутри вашего Си файла. Вы можете, конечно, использовать исходные ассемблерные файлы как часть вашего проекта также. Синтаксис встроенного ассемблерного кода следующий:

```
asm("<string>");
```

Множественные операторы ассемблера могут отделяться символом новой строки \n. Можно использовать конкатенации строк, чтобы определить множественные операторы без использования дополнительных ключевых слов asm. Чтобы обращаться к переменной Си из оператора ассемблера, используйте формат %<name>:

```
register unsigned char val;

asm("mov %uc, R10\n"
    "sleep\n");
```

Этим способом можно сослаться на любой символ Си, исключая метки оператора Си goto. Вообще, использование встроенного ассемблерного кода для ссылок на локальные регистры ограничено. Возможно, что никакие регистры не будут являться доступными, если вы объявили слишком много регистровых переменных в функции. В таком случае, вы получили бы ошибку ассемблера. Также не существует способа управления размещением регистровых переменных, что может привести к ошибке встроенного кода.

Встроенный ассемблер может использоваться внутри или вне функции Си. Компилятор выравнивает каждую строку встроенного ассемблера для удобочитаемости. Вы можете получить предупреждение об операторах asm, которые появляются вне тела функции. Вы можете игнорировать эти предупреждения.

8.4. Регистры ввода/вывода

Так как MSP430 имеет архитектуру фон-Неймана с памятью RAM, ROM и регистрами ввода/вывода, расположенными в одном адресном пространстве, регистры ввода/вывода адресуются, используя обычный механизм Си для доступа к памяти. Например:

```
#define IE1 (*(volatile unsigned char *)0x0000)
...
... = IE1;     // read from location 0
IE1 = ... ;    // write to location 0
```

Квалификатор volatile сообщает компилятору, что все способы доступа к памяти должны быть доступны. Иначе говоря, правила Си позволяют кэшируемым значениям использоваться многократно. Заголовочные файлы `msp430x???x.h` определяют регистры ввода/вывода согласно документации TI.

8.5. Абсолютная адресация памяти

Ваша программа может иметь необходимость адресовать абсолютные области памяти. Например, внешние периферийные устройства обычно отображаются на специфические адреса памяти. Они могут включать интерфейс с LCD и двухпортовой SRAM. В настоящее время вы можете использовать встроенный ассемблер или отдельный ассемблерный файл для объявления данных, расположенных по определенным адресам памяти. В более поздних версиях компилятора мы можем реализовать эту возможность в Си.

В следующих примерах, предполагается, что имеются двухбайтовый регистр управления LCD по адресу `0x1000`, двухбайтовый регистр данных LCD по следующему адресу (`0x1002`), и также имеется 100-байтовая двухпортовая память SRAM, расположенная по адресу `0x2000`.

8.5.1. Использование ассемблерного модуля

В файле ассемблера поместите следующее:

```
.area memory(abs)
.org 0x1000
    _LCD_control_register:: .blkw 1
    _LCD_data_register::    .blkw 1
.org 0x2000
    _dual_port_SRAM::       .blkb 100
```

В вашем файле Си, вы должны объявить их так:

```
extern unsigned int LCD_control_register, LCD_data_register;
extern char dual_port_SRAM[100];
```

Обратите внимание на соглашение интерфейса о добавлении именам внешних переменных префикса '_' в файле ассемблера и использовании двух двоеточий, чтобы определить их как глобальные переменные.

8.5.2. Использование встроенного ассемблера

Встроенный ассемблер использует тот же синтаксис регулярного ассемблера, за исключением того, что его команды заключены в псевдофункцию `asm()`. В файле Си, предыдущий ассемблерный код становится следующим:

```
asm(".area memory(abs)\n"
    ".org 0x1000\n"
    "_LCD_control_register:: .blkw 1\n"
    "_LCD_data_register::    .blkw 1");
asm(".org 0x2000\n"
    "_dual_port_SRAM::       .blkb 100");
```

Обратите внимание на использование `\n`, для разделения строк. В файле Си вы все еще должны объявить переменные как "`extern`" (как в предшествующем примере), точно так же, как в случае использования отдельного файла ассемблера, так как компилятор Си в действительности не знает того, что находится внутри операторов `asm`.

8.6. Си-задачи

Как описано в разделе 9.2. Интерфейс ассемблера и соглашения о вызовах, компилятор обычно генерирует код, чтобы сохранять и восстанавливать предохраняемые регистры. В некоторых случаях, это поведение может быть нежелательно. Например, если вы используете RTOS (Real Time Operating System), она сама управляет сохранением и восстановлением регистров как частью процесса переключения задач, и код, вставленный компилятором, становится избыточным.

Чтобы отключить это поведение, используйте “`#pragma ctask`”. Например:

```
#pragma ctask drive_motor emit_siren
....
void drive_motor() { ... }
void emit_siren() {...}
```

Прагма должна появляться перед определениями функций. Обратите внимание, что по умолчанию, процедура “`main`” имеет этот набор атрибутов, так как `main` никогда не должна возвратиться, и для нее не нужно сохранять и восстанавливать никакие регистры.

8.6.1. Мониторы

В некоторых RTOS полезно вызывать функцию, запрещающую прерывания при работе в какой-либо критической области. В идеале, при возврате функция должна восстанавливать состояние флага глобального разрешения прерываний GIE (Global Interrupt Enable) в его предыдущее состояние. ICC430 имеет прямую поддержку определения таких функций простым использованием `#pragma monitor`. Например:

```
#pragma monitor scheduler
```

объявляет `scheduler` как функцию монитора. Прагма должна появляться перед определениями функций. Компилятор сгенерирует команды:

```
push R2
dint
```

на входе в функцию, сохраняя предыдущее состояние регистра состояния Status Register (`SR` или `R2`), и затем запрещая прерывания инструкцией `dint`. На выходе из такой функции генерируется инструкция `reti` (возврат из прерывания) вместо инструкции `ret`. Это дает желаемый эффект восстанавливая `SR`/`R2` перед возвратом. Функция монитора следует обычным соглашениям по вызову.

8.7. Обработка прерываний

8.7.1. Си обработчики прерываний

Обработчики прерываний можно писать на Си. Перед определением функции вы должны сообщить компилятору, что она является обработчиком прерывания, используя прагму:

```
#pragma interrupt_handler <func name>:<vector offset> *
```

“vector offset” – смещение вектора является смещением адреса прерывания относительно начала базы векторов, определяемой архитектурой контроллера и расположенной по адресам от 0xFFE0 до 0xFFFF. Смещения векторов определены в специфических для устройств файлах msp430x???.h. Эта прагма имеет два эффекта:

- Для функции обработки прерываний, компилятор генерирует команду reti вместо ret, и сохраняет и восстанавливает все регистры, используемые в функции.
- Компилятор генерирует вектор прерывания на основе смещения вектора.

Например:

```
#include <msp430x14x.h>
...
#pragma interrupt_handler timer_handler:TIMERA1_VECTOR
...
void timer_handler()
    {
    ...
    }
```

Вы можете поместить несколько имен в одну прагму interrupt_handler, разделяя их пробелами. Если вы желаете использовать один обработчик прерывания для нескольких векторов, объявите его несколько раз с различными смещениями векторов. Например:

```
#pragma interrupt_handler timer_ovf:TIMERA0_VECTOR
timer_ovf:TIMERA1_VECTOR
```

8.7.2. Модифицирование сохраненного регистра SR

Иногда полезно модифицировать сохраненный в стеке регистр состояния SR в обработчике прерывания, для того чтобы выполнение программы могло бы продолжиться в другом режиме. Вы можете использовать две внутренние функции – _BIS_SR_IRQ(int) и _BIC_SR_IRQ(int) для решения этой задачи. Например:

```
_BIS_SR_IRQ(0x10);    // turn off CPU, CPUOff
```

Эти "функции" объявлены в msp430def.h. Это не истинные функции, они известны компилятору и он генерирует непосредственные инструкции для модифицирования SR.

8.7.3. Ассемблерные обработчики прерываний

Можно написать обработчик прерывания на ассемблере. Однако если вы вызываете функции Си из ассемблерного обработчика, ассемблерная процедура должна сохранять и восстанавливать volatile регистры (см. 9.2. Интерфейс ассемблера и соглашения о вызовах) так как функции Си этого не делают (если они не объявлены как процедуры обработки прерывания, но тогда они не должны вызываться непосредственно).

При использовании ассемблерных обработчиков прерывания, векторы должны определить вы. Используйте атрибут “abs”, чтобы объявить абсолютную область (см. 12.4. Директивы ассемблера) и используйте директиву “.org”, чтобы определить правильный адрес.

```
.area vector(abs) ; interrupt vectors
.org 0xFFE2
.word _port2_handler
```

9. АРХИТЕКТУРА ВРЕМЕНИ ИСПОЛНЕНИЯ

9.1. Размеры типов данных

Тип	Размер (байт)	Диапазон
unsigned char	1	0..255
signed char	1	-128..127
char (*)	1	0..255
unsigned short	2	0..65535
(signed) short	2	-32768..32767
unsigned int	2	0..65535
(signed) int	2	-32768..32767
pointer	2	0..65535
unsigned long	4	0..4294967295
(signed) long	4	-2147483648..2147483647
float	4	-1.175e-38..3.40e+38
double	4	-1.175e-38..3.40e+38

(*) Тип `char` эквивалентен типу `unsigned char`.

Типы `float` и `double` используют 32-х битный формат стандарта IEEE с 8-битной экспонентой, 23-х битной мантиссой, и 1-битным знаковым разрядом.

Типы битовых полей могут быть как знаковыми, так и беззнаковыми, но они будут упакованы в минимальное пространство. Например:

```
struct {
    unsigned a : 1, b : 1;
};
```

Размер данной структуры всего 1 байт. Битовые поля упаковываются справа налево.

9.2. Интерфейс ассемблера и соглашения о вызовах

9.2.1. Внешние имена

Внешние имена Си добавляют себе префикс в виде знака подчеркивания. Например, функция `main` будет именоваться `_main`, если на нее ссылаться из ассемблерного модуля. Идентификаторы имеют длину 32 значащих символа. Чтобы сделать ассемблерный объект глобальным, используйте два двоеточия после имени. Например:

```
_foo::
    .word 1
```

в файле Си соответствует следующему объявлению:

```
extern int foo;
```

9.2.2. Регистры аргументов и возвращаемых значений

При отсутствии прототипа функции, целочисленные параметры меньшие, чем `int` (например, `char`) должны повышаться до типа `int`. Если прототип функции доступен, стандарт Си оставляет решение реализации компилятора. ICCV7 для MSP430 не повышает тип параметра, если прототип функции доступен. При вызове функции с переменными параметрами используя синтаксис `stdarg` Стандартного Си:

```
int printf(char *, ...);
```

все параметры передаются через стек. В противном случае, первые 4 байта параметров передаются через регистровую пару R14/R15. Следующие 4 байта передаются через пару R12/R13. Все остальные параметры передаются через стек.

Целочисленные параметры меньшие, чем `int` (например, `char`) не расширяются до `int`, если прототип функции доступен. Длинные и с плавающей точкой параметры передаются только через "четные" регистровые пары (или через стек). Например:

```
void foo(int i, long l);
```

параметр `i` передается через R14, регистр R15 не используется, а параметр `l` передается через пару R12/R13.

Целые значения возвращаются в R14, а длинные и плавающие возвращаются в R14/R15. Байтовые результаты не расширяются до целых.

9.2.3. Предохраняемые регистры

Ассемблерные функции должны сохранять и восстанавливать следующие регистры:

- R4/R5/R6/R7/R8/R9

Эти регистры называются предохраняемые регистры, так как их содержимое не изменяется при вызове функции. Компилятор использует эти регистры для размещения локальных переменных.

9.2.4. Volatile регистры

Другие регистры:

- R10/R11/R12/R13/R14/R15

могут использоваться функцией без сохранения и восстановления. Эти регистры называются volatile регистры, т.к. их содержимое может изменяться при вызове функции.

9.2.5. Обработчики прерываний

Обратите внимание, что в отличие от нормальной функции, обработчик прерывания должен сохранять и восстанавливать все регистры, которые он использует. Это выполняется автоматически, если вы используете возможности компилятора и объявляете функцию Си как обработчик прерывания. Если вы пишете обработчик прерывания на ассемблере, и если он вызывает нормальные функции Си, то ассемблерный обработчик должен сохранять и восстанавливать volatile регистры, так как нормальные функции Си не предохраняют их. Так как прерывания функционируют асинхронно по отношению к нормальным операциям программы, обработчик прерывания и функции, которые он вызывает, не должны изменять никакие машинные регистры.

9.3. Функции, возвращающие нецелые значения

Всегда используйте прототип функции прежде, чем вы вызвать функцию, так как передаваемые параметры и возвращаемые значения находятся в разных местах в зависимости от типов данных параметров или значения, возвращаемого функцией. Например, вы должны включать `(#include)` заголовочный файл `<math.h>` перед вызовом любой функции с плавающей точкой. Иначе, ваша программа не будет работать.

9.3.1. Возвращаемые значения типов Long и Float

Значения типов `long` и `float` возвращаются функцией в R14/R15.

9.3.2. Передача структуры по значению

При передаче по значению, структура всегда передается через стек, а не в регистрах. Передача структуры по ссылке (то есть передача адреса структуры) происходит так же, как при передаче адреса любого элемента данных, то есть передается указатель на структуру (который является 2 байтами).

9.3.3. Возврат структуры по значению

Когда вызывается функция, возвращающая структуру, вызывающая функция размещает временную память и передает скрытый указатель на нее в вызываемую функцию. При возврате из такой функции, возвращаемое значение копируется в эту временную память.

9.4. Машинные процедуры Си

Большинство операций Си транслируется в прямые машинные инструкции. Однако, некоторые операции, транслируются в вызовы процедур, потому что они включают много машинных команд и вызвали бы слишком большое разрастание кода, если бы транслировались в последовательность встроенных команд. Эти процедуры написаны на ассемблере и могут отличаться тем, что их имена не начинаются с подчеркивания. Вот некоторые из часто встречаемых процедур со следующими префиксами:

- `lsr16, lsr32 ...` – выполняют операции сдвига 16- и 32-разрядных данных
- `mpy, div, mod ...` – 32-разрядные процедуры для длинных и плавающих чисел.

9.5. Области программы

Компилятор генерирует код и данные в различных областях. См. 12.4. Директивы ассемблера. Области, используемые компилятором, упорядочены здесь по возрастанию адресов памяти:

9.5.1. Память только для чтения

- `func_lit` – область таблиц функций. Каждое слово в этой области содержит адрес входа в функцию. Для полной совместимости с 12.1. Компрессор кода, все косвенные ссылки на функции должны иметь дополнительный уровень косвенности. Это выполняется автоматически в Си, если вы вызываете функцию по указателю. В ассемблере это иллюстрируется следующим примером:

```
; assume _foo is the name of the function
.area func_lit
PL_foo:: .word _foo     ; create a function table entry
.area text
call PL_foo
```

  Так как MSP430 может использовать любой допустимый режим адресации в инструкции `call`, дополнительный уровень косвенности дает минимальные накладные расходы.

- `idata` – в этой области хранятся начальные значения глобальных данных и строк.
- `interrupt vectors` – эта область содержит векторы прерывания.
- `lit` – эта область содержит целочисленные и плавающие константы, и т.п.
- `text` – эта область содержит код программы.

9.5.2. Память данных

- `data` – это область данных, содержащая глобальные и статические переменные и строки. Начальные значения глобальных переменных и строк хранятся в области `idata` и копируются в область данных при старте программы.
- `bss` – это область данных, содержащая неинициализированные глобальные переменные Си. По стандарту ANSI C, эти переменные инициализируются нулями при старте программы.

Задача компоновщика состоит в том, чтобы собрать все области совпадающих типов из всех входных объектных файлов и объединить их вместе в выходном файле. См. 12.6. Операции компоновщика.

9.5.3. Области, определяемые программистом

В большинстве случаев вы не должны определять точное расположение отдельных элементов данных. Например, если вы имеете глобальную переменную, она будет расположена где-нибудь в области данных.

Однако имеются случаи, когда вы можете захотеть определить точное местоположение элемента данных или группы данных:

- батарейная SRAM, двухпортовая SRAM, и т.п. – иногда подобные объекты необходимо расположить в специальных областях памяти RAM.

Существует два пути решения этого вопроса:

1. переместимая область – в ассемблерном модуле вы можете создать новую область программы, и затем определить начальный адрес в окне редактирования *Advanced>Other Options* в 4.13. Опции компилятора: Целевое устройство. Например, в ассемблерном файле:

```
.area battery_sram
_var1:: .blkw 1          ; note _ in the front
_var2:: .blkb 1          ; and two colons
```

В Си эти переменные должны быть объявлены так:

```
extern int  var1;
extern char var2;
```

Пусть батарейная SRAM начинается с адреса `0x4000`. В окне редактирования *Advanced>Other Options* введите:

```
-bbattery_sram:0x4000
```

См. в 4.13. Опции компилятора: Целевое устройство описание спецификатора адреса.

2. абсолютная область – вы можете также определять области программы, имеющие абсолютные начальные адреса без необходимости определения адреса используя ключей командной строки компоновщика. Например, для того же самого примера, вы можете записать в ассемблерном файле следующее:

```
.area battery_sram(abs)
.org 0x4000
_var1:: .blkw 1          ; note _ in the front
_var2:: .blkb 1          ; and two colons
```

Атрибут `(abs)` сообщает ассемблеру, что область не нуждается в настройке и допускается использование директивы `.org`. В этом примере `.org` используется для установки начального адреса. На Си объявление будет точно таким же, как и раньше.

Если вы имеете инициализированные данные, вы можете также использовать прагму в тексте Си, чтобы определить эти данные (заметим, что это работает только с инициализированными данными):

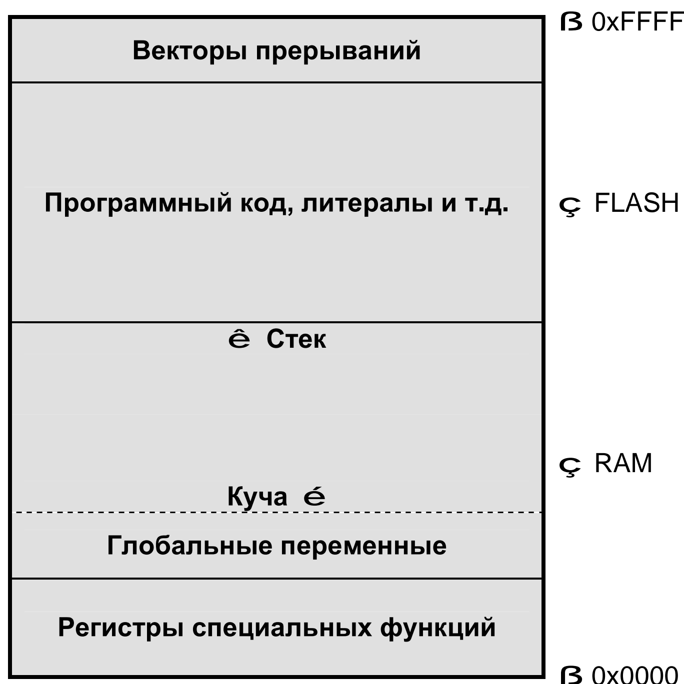
```
#pragma abs_address:0x4000
int var1   = 5;
char var2 = 0;
#pragma end_abs_address
```

9.6. Функции стека и кучи

Кроме статических областей программы, среда времени исполнения Си содержит две дополнительных области данных – стек и куча. Стек используется для вызовов процедур, локальных и временных переменных, и передачи параметров. Куча используется для динамически размещаемых объектов, создаваемых стандартными функциями Си `malloc()`, `calloc()` и `realloc()`.

Не имеется никаких средств для проверки переполнения стека, поэтому вы должны использовать его осторожно. Например, серия рекурсивных вызовов с большим количеством локальных переменных быстро поглощает пространство стека. Когда стек накладывается на другие данные или сохраненные адреса, программа потерпит аварию. Стек растет вниз и инициализируется верхним адресом данных SRAM в 7.2. Файл запуска.

Чтобы использовать функции кучи, вы должны сначала инициализировать область кучи. См. 7.7. Стандартная библиотека и функции памяти.



Если вы используете `#pragma text / data / lit / abs_address` чтобы назначить собственные области памяти, вы должны самостоятельно гарантировать, что их адреса не накладываются на адреса, используемые компоновщиком. При наложении адресов компоновщик не всегда может генерировать ошибку, поэтому вы всегда должны проверять файл карты `.mp`. Используйте меню (*View>Map File*) для поиска потенциальных проблем.

10. ОТЛАДКА

10.1. Общие приемы отладки

Отладка встроенных программ может быть очень затруднительна. Если ваша программа не выполняется, как ожидалось, это может происходить из-за одной или нескольких следующих причин.

- Конфигурации по умолчанию у некоторых процессоров могут быть не теми, которые ожидаются, исходя из логики пользователя. Некоторые примеры:
  - Фирма Atmel для процессора Mega128 оставила фабричные установки конфигурации такими, чтобы процессор вел себя подобно старому устройству M103C с его набором уставок. Если вы запросите компилятор генерировать код для устройства M128, то оно не будет работать, так как режим совместимости с M103 использует другую карту памяти для внутренней SRAM. Эта "мелкая деталь", вероятно, составляет большинство запросов поддержки, которые мы получаем от пользователей M128.
  - В Atmel AVR, по умолчанию некоторые из Mega устройств используют внутренние тактовые генераторы вместо внешних.
  - В устройствах Freescale HC12/S12, по умолчанию сторожевой таймер активен и должен быть заблокирован в течение первых 64 циклов после сброса, если вы желаете дезактивировать его.
  - В устройствах Freescale HCS12, расположение вектора сброса и других векторов прерывания зависит от того, имеет ли устройство или плата бортовой монитор, или вы используете устройство BDM и т.д.
  - Устройства ARM7 от различных изготовителей имеют очень разные контроллеры прерываний.
  - Для устройств с внешней памятью SRAM, аппаратный интерфейс может нуждаться во времени, чтобы стабилизироваться после сброса устройства, прежде, чем к внешнему SRAM можно корректно обращаться.
- Ваша программа должна использовать правильные адреса памяти и систему команд. Различные устройства в том же самом семействе могут иметь различные адреса памяти или могут даже иметь несколько различные системы команд (например, некоторые устройства могут иметь аппаратную команду умножения). Наша среда разработки обычно обрабатывает эти детали за вас. Когда вы выбираете устройство по имени, среда генерирует подходящие ключи транслятора и компоновщика. Однако если ваша аппаратура несколько отличается (например, вы можете иметь внешнюю память SRAM) или, если устройство, которое вы используете, еще не поддержано средой разработки явно, все же, вы можете обычно выбирать ваше устройство как “Custom” и вводить данные вручную.
- Ваша программа может содержать логические или другие ошибки программирования. Наиболее трудные для отслеживания ошибки – это наложение записи в память, где данные изменяются неумышленно. Например, вы можете иметь переменную указатель, указывающую на недопустимый адрес, и запись через переменную указатель может иметь катастрофические последствия, не обнаруживаемые немедленно. Другой источник таких ошибок памяти – переполнение стека. Стек обычно использует пространство совместно с переменными SRAM и если стек переполняется в область переменных, случаются неприятности.

- Ложное или неожиданное поведение прерываний могут разрушить вашу программу:
  - Вы должны всегда устанавливать обработчик для “неиспользуемых” прерываний. Непредвиденное прерывание может вызвать проблемы.
  - Осторожно обращайтесь к переменным с размером большим, чем естественный размер данных процессора, которые требуют нескольких циклов доступа. Например, запись 16-разрядного значения в Atmel AVR требует, по крайней мере, двух команд. Следовательно, обращение к переменной из основной прикладной программы и программы обработки прерывания должно быть выполнено с осторожностью. Например, основная программа, записывающая 16-разрядную переменную, может быть прервана в середине последовательности 2-х команд. Если программа обработки прерывания проверяет значение переменной, переменная может быть в неоднозначном состоянии.
  - Большинство архитектур процессоров не позволяет вложенные прерывания по умолчанию. Если вы обходите механизм процессора и используете вложенные прерывания, будьте внимательны, чтобы не получить несогласованные вложенные прерывания.
  - В большинстве систем лучшим решением является обработка прерывания с такой высокой скоростью, и используя такой минимум ресурсов, насколько возможно. Вы должны быть внимательны при вызове функций (собственных или библиотечных) внутри программы обработки прерывания. Например, почти всегда является плохой идеей вызывать такую сверхтяжелую библиотечную функцию как `printf` внутри программы обработки прерывания.
  - За некоторыми исключениями, наши трансляторы генерируют реентерабельный код. То есть ваша функция может быть прервана и вызвана снова, пока вы осторожны с глобальными переменными. Большинство библиотечных функций также реентерабельны, исключая printf и связанные с ней функции, являющиеся основными исключениями. В компиляторах Freescale HC11 и HC12/S12 выражения с плавающей запятой нереентерабельны, так как низкоуровневые функции с плавающей точкой используют глобальные переменные как “регистры”.
- Тщательно проверяйте интерфейс с внешней памятью. Например, делайте не только проход по всей внешней RAM, но и проверку записи нескольких шаблонов в одиночном цикле, поскольку может случиться, что старшие биты адреса работают неправильно.
- Компилятор может делать непредвиденные действия даже притом, что это правильно. Например, для RISC-подобных процессоров типа Atmel AVR, TI MSP430 и ARM CPU, компиляторы могут помещать разные локальные переменные в тот же самый машинный регистр, пока использование локальных переменных не накладывается. Это значительно улучшает сгенерированный код, даже притом, что это может удивлять при отладке. Например, если вы помещаете окно часов в две переменные, которые компилятором размещаются в одном и том же регистре, обе переменные изменялись бы даже притом, что ваша программа изменяет только одну из них.
- Машинно-независимый оптимизатор делает отладку даже более спорной. MIO может устранять или перемещать код или изменять выражения, а для RISC-подобных процессоров, распределитель регистров может назначать различные регистры или адреса памяти одной и той же переменной в зависимости от места использования. К сожалению, в настоящее время большинство отладчиков имеют только ограниченную поддержку отладки оптимизированного кода.

- Вы можете столкнуться с ошибкой компилятора. Если вы сталкиваетесь с сообщением об ошибке в форме

  ```
  "Internal Error! ...,"
  ```

  это означает, что транслятор обнаружил внутреннее противоречие. Если вы видите сообщение в форме

  ```
  ...The system cannot execute <one of the compiler programs>
  ```

  это означает, что, к сожалению, транслятор потерпел крах при обработке вашего кода. В любом случае, вы будете должны связаться с нами. См. 1.7. Поддержка.

- Вы можете столкнуться с ошибкой компилятора. К сожалению, компилятор – набор относительно сложных программ, которые вероятно содержат ошибки. Наш внешний интерфейс (часть, которая делает синтаксический и семантический анализ входных программ Си) является особенно выверенным, поскольку мы лицензируем LCC программное обеспечение по высоко уважаемому внешнему интерфейсу компилятора ANSI C. Мы проверяем наши трансляторы полностью, включая почти исчерпывающее тестирование базовых операций всех поддерживаемых целочисленных операторов и типов данных.

  Однако, несмотря на все наше тестирование, компилятор может все еще генерировать неправильный код. Вероятность этого очень низка, поскольку большинство проблем поддержки не относится к ошибкам компилятора, даже если заказчик уверен в этом. Если вы думаете, что нашли проблему компилятора, то всегда помогает, если вы попробуете упростить вашу программу или функцию так, чтобы мы смогли дублировать ее. См. 1.7. Поддержка.

10.1.1. Тестирование логики программы

Так как компилятор использует стандарт ANSI C, общий метод разработки программ состоит в использовании компилятора для PC, такого, как **Borland C** или **Visual C**, для первоначальной отладки логики вашей программы, компилируя ее как программу для PC. Очевидно, что аппаратно зависимые части должны быть изолированы и заменены или подменены процедурами-заглушками. Обычно, используя этот метод, можно отладить 95 % кода вашей программы и даже больше.

Если ваша программа работает неверно, наблюдается непорядок с переменными, принимающими странные значения, или счетчик команд адресует неожиданные места, возможно, в вашей программе при записи в память происходит перекрытие участков памяти. Необходимо убедиться, что переменные-указатели ссылаются на допустимые участки памяти и что стек не записывается поверх памяти данных.

10.1.2. Файл листинга

Один из выходных файлов, произведенных компилятором – файл листинга по имени `<file>.lst`. Файл листинга содержит ассемблерный код вашей программы, сгенерированный компилятором, с включениями исходного текста Си, машинным кодом и адресами. Значения данных не содержатся в данном файле, а библиотечный код показывается только в зарегистрированной версии.

10.2. Отладка в NoICE430

NoICE430 – версия популярного отладчика NoICE (http://www.noicedebugger.com) написанный для TI MSP430. Он поддерживает программирование FLASH памяти устройств MSP430, включая дешевый комплект TI FET. Обеспечивается полная отладка на уровне Си и ассемблера без ограничений на контрольные точки, наблюдение за переменными, дампы памяти, и т.д.

NoICE430 распознает отладочный формат ICC430 (файлы с расширением `.dbg`). После того как вы скомпилировали проект, вы можете щелкнуть на кнопке NoICE430 на панели инструментов ICC430 или вызвать *Tools>NoICE430* и запустить отладчик.

Вы можете загрузить `.DBG` файл, используя меню *File>Load*. Вы можете также загрузить Intel HEX файл, если нужно только запрограммировать устройство. Конечно, среда ICC430 IDE включает встроенный программатор FLASH и NoICE430 для этой цели не обязателен.

Вы можете выбрать *Run>"go until main after load"*. Отладчик выполнит вашу программу до функции `main`. Иначе, он начал бы с кода запуска. Вы можете переключать режим отображения кода Си или ассемблера. Доступны также всесторонняя справка и советы.

Заметьте, что только версия PROFESSIONAL компилятораICC430 выдает отладочную информацию о структурах. С версией STANDARD вы не сможете увидеть члены структур.

10.2.1. Символы портов

Чтобы использовать символы портов, такие как `P1IN`, `P1OUT`, `P1DIR` и т.д., вы должны определить эти символы с адресами и типами данных, чтобы NoICE430 мог их распознать. Если вы имеете файл `noice430.noi` в каталоге `\icc\bin`, NoICE430 загрузит этот файл перед выполнением вашей программы, чтобы вы могли помещать в него определения символов портов. Вы можете использовать программу командной строки `msp430equates.exe` в `\icc\bin`, чтобы генерировать `.noi` файл из `.h` файла включения. Мы обеспечиваем файл по умолчанию `noice430.noi`, который преобразован из файла `msp430x14x.h`, пригодный для устройств F14x, таких как F149. Если вы используете другое устройство, вы должны выполнить преобразование самостоятельно. Запустите эту программу без параметров для получения справки.

Вы можете иметь несколько файлов определения символов. Вы можете использовать любой такой файл, используя команду NoICE430 *File>Play*.

11. КОМПИЛЯТОР КОМАНДНОЙ СТРОКИ

11.1. Процесс компиляции

Под дружественной средой разработки находится набор программ компилятора командной строки. Вы не обязаны разбираться в этом материале, чтобы использовать компилятор, поэтому данная глава предназначена для тех, кто интересуется внутренними процессами.

С данным списком файлов проекта, работа компилятора заключается в трансляции файлов в исполнимый файл в некотором выходном формате. Обычно, процесс трансляции скрыт от вас с помощью менеджера проекта. Однако может оказаться важным иметь представление о том, что происходит внутри:

1. Компилятор компилирует каждый исходный файл Си в ассемблерный файл.
2. Ассемблер транслирует каждый ассемблерный файл (после компилятора или ассемблерный файл, который вы написали сами) в перемещаемый объектный файл.
3. После трансляции всех файлов в объектные файлы, компоновщик объединяет их вместе для получения исполнимого файла. Кроме того, также выводятся файл карты, файл листинга и отладочные информационные файлы.

Все эти шаги поддерживаются драйвером компилятора. Вы передаете ему список файлов и запрашиваете компиляцию их в исполнимый файл (по умолчанию) или в некоторый промежуточный формат (например, в объектные файлы). Драйвер вызывает компилятор, ассемблер и при необходимости компоновщик.

Фактически, среда разработки, даже не связывается с помощью интерфейса с драйвером компилятора непосредственно. Она генерирует make-файл и вызывает программу `make` для интерпретации make-файла, которая и вызывает драйвер компилятора.

11.2. Утилита Make

Утилита make (imakew) является подмножеством стандартной make программы Unix. Она читает входной файл, содержащий список зависимостей и ассоциированных действий, чтобы определить временные зависимости. Формат в общем случае состоит из имени обрабатываемого файла, сопровождаемого списком файлов, от которых он зависит, с последующей группой команд для модификации файла на основе этих зависимостей:

```
target: dependence1 dependence2 ...
<TAB>action1
<TAB>action2
...
```

Символ табуляции важен для некоторых действий. Утилите make не нравится, если вы используете пробелы вместо символа табуляции. Каждый зависимый файл может быть целью в make-файле. Сопровождение каждого зависимого файла выполняется рекурсивно перед попыткой поддержки текущего целевого файла. Если, после обработки всех зависимостей, целевой файл оказался отсутствующим или более старым, чем любой из файлов зависимостей, make выполняет прилагаемые команды или неявным образом перекомпилирует его.

Входной файл по умолчанию – это makefile, но вы можете изменить это опцией командной строки -f <filename>. Если целевой файл не определен в командной строке, make использует первый целевой файл, определенный в makefile.

Такое применение make достаточно для использования в среде разработки. Однако, если вы – опытный пользователь, нуждающийся в полной мощности утилиты make, вы должны использовать полнофункциональную реализацию программы make типа GNU make.

11.2.1. Параметры утилиты Make

- -f <makefile> – использовать указанный файл вместо makefile по умолчанию.
- -h – вывести короткое справочное сообщение.
- -i – игнорировать коды ошибок, возвращенные командами. Обычное поведение состоит в том, что make останавливается, если команда возвращает код ошибки.
- -n – режим невыполнения. Вывести команды, но не выполнять их.
- -p – печатать все макросы и имена целевых файлов.
- -q – make возвращает 1, если целевой файл устарел. Иначе возвращается 0.
- -s – режим молчания. Не печатать командные строки перед их выполнением.
- -t – относится больше к целевым файлам (обеспечение их обновления), чем к выполнению команд.

Вы можете также определить макрос make в командной строке определением:

```
macro=value
```

11.3. Драйвер

Драйвер компилятора исследует каждый входной файл и действует на основе расширения файла и полученных параметров командной строки. Файлы с расширениями .c и .s являются исходными файлами Си и ассемблера соответственно. Философия разработки в среде состоит в том, чтобы сделать ее настолько легкой в использовании насколько возможно. Компилятор командной строки, тем не менее, является чрезвычайно гибким. Вы управляете его поведением, передавая ему параметры командной строки. Если вы хотите связать компилятор с помощью интерфейса с вашей собственной оболочкой (например, Codewright или редактор Multiedit), вы должны разрешить несколько вопросов:

- Сообщения об ошибках в исходных файлах начинаются с `!E file(line):...` Предупреждения используют тот же самый формат, но используют префикс `!W` вместо `!E`.
- Чтобы обойти ограничения на длину командной строки в Windows 95/NT, вы можете поместить параметры командной строки в файл, и передать его компилятору как `@file` или `@-file`. Если вы передаете его как `@-file`, компилятор удалит `file` после выполнения.

11.4. Параметры компилятора

Среда разработки управляет поведением компилятора, передавая параметры командной строки драйверу компилятора. Обычно вы не должны знать, что делают параметры командной строки, но вы можете видеть их в сгенерированном make-файле и в окне состояния, когда выполняете компиляцию. Тем не менее, эти страницы описывают опции, используемые средой на случай, если вы захотите управлять компилятором, используя ваш собственный редактор-среду типа Codewright. Все параметры передаются драйверу, и драйвер в свою очередь передает соответствующие параметры дальше.

Общий формат команды следующий:

```
icc430 [ command line arguments ] <file1> <file2> ... [ <lib1> ... ]
```

Где `icc430` – это имя драйвера компилятора. Как можно видеть, вы можете вызывать драйвер с множеством файлов, и он выполнит операции со всеми файлами. По умолчанию драйвер свяжет все объектные файлы вместе для создания выходного файла.

Драйвер автоматически добавляет `-I<install root>\include` к параметрам препроцессора Си и `-L<install root>\lib` к параметрам компоновщика.

Для большинства общих опций, драйвер знает, какие параметры для какого прохода компилятора предназначены. Вы можете также определять, к какому проходу применяется параметр, используя префикс `-W<c>`. Например:

- `-Wp` – препроцессор. Например, `-Wp-e`.
- `-Wf` – компилятор. Например, `-Wf-hwmult`.
- `-Wa` – ассемблер.
- `-Wl` – (буква el) - компоновщик.

11.4.1. Параметры драйвера

- `-c` – только компиляция файла в объектный файл (компоновщик не вызывается).
- `-o <name>` – имя выходного файла. По умолчанию имя выходного файла совпадает с именем входного файла или с именем первого файла, если вы передаете драйверу список входных файлов.
- `-v` – подробный режим. Распечатывается каждый проход компилятора, по мере выполнения.

11.4.2. Параметры препроцессора

- `-D<name>[=value]` – определяет макрос. См. 4.12. Опции компилятора: Компилятор. Драйвер и среда разработки предопределяют некоторые макросы. См. 5.2. Предопределенные макросы.
- `-U<name>` – отменяет определение макроса.
- `-e` – допускает комментарии C++.
- `-I<dir>` – (заглавная буква `I`) Определяет места поиска заголовочных файлов. Может быть несколько параметров `-I`.

11.4.3. Параметры компилятора

- `-e` – допускать расширения, включая двоичные константы 0b????.
- `-l` (буква el) – генерировать файл листинга.
- `-A -A` (два `-A`) – включить строгую проверку ANSI. Одиночная `-A` включает частичную проверку ANSI.
- `-g` – генерировать отладочную информацию.

При использовании с драйвером, следующая опция должна использовать префикс `-Wf`, например `-Wf-hwmult`.

- `-hwmult` – использовать модуль аппаратного умножения.

11.4.4. Параметры ассемблера

- `-n` – вообще используется только для ассемблирования файла запуска (См. 7.2. Файл запуска). Обычно ассемблер неявно вставляет `.area text` в начале обрабатываемого файла. Это позволяет в общем случае опускать директивы области в начале модуля кода для корректного ассемблирования. Однако файл запуска имеет специальные требования, чтобы эта неявная вставка не выполнялась.

11.4.5. Параметры компоновщика

- `-L<dir>` – определить библиотечный каталог. Может быть определено множество каталогов, и их поиск ведется в обратном порядке (последний определенный каталог ищется первым).
- `-O` – вызывать компрессор кода (работает только в ПРОФЕССИОНАЛЬНОЙ версии).
- `-m` – генерировать файл карты.
- `-g` – генерировать информацию для отладки. Файл отладки имеет расширение .DBG.
- `-u<crt>` – использовать `<crt>` вместо файла старта. Если файл задан только именем без информации о пути, то он должен быть размещен в библиотечном каталоге.
- `-fintelhex` – выходным форматом является `Intel HEX`.
- `-dram_end:<address>` – определяет конец области памяти RAM. Используется файлом запуска (См. 7.2. Файл запуска) для инициализации указателя стека. Обычно совпадает с концом области данных.
- `-l<libname>` – компоновать со специфицированными библиотечными файлами в дополнение с библиотекой по умолчанию `libc430.a`. Может использоваться для изменения поведения функции в `libc430.a`, т.к. `libc430.a` всегда связывается последним. `libname` – имя библиотечного файла без префикса `lib` и без суффикса `.a`. Например:

```
-llp430   "liblp430.a"   using full printf
-lfp430   "libfp430.a"   using floating point printf
```

- `-F<pat>` – заполнять неиспользуемые области памяти ROM шаблоном `pat`. Шаблон должен быть целым числом. Используйте префикс 0x для шестнадцатеричного числа.
- `-R` – не компоновать с файлом старта или с библиотечным файлом по умолчанию. Полезно, если вы пишете чисто ассемблерное приложение.

Определения адресов

Если вы используете `#pragma text / data / lit / abs_address` , чтобы назначить ваши собственные области памяти, вы должны сами гарантировать, что их адреса не накладываются на адреса, используемые компоновщиком. В случае наложения памяти компоновщик может не сгенерировать ошибку, поэтому вы всегда должны сами проверять файл карты `.mp` (используйте меню *View->Map File*) для поиска потенциальных проблем.

- `-b<area>:<address ranges>` – назначение адресных интервалов для области. Вы можете использовать это, чтобы создавать ваши собственные области с собственными адресами. См. 9.5. Области программы. Формат задания диапазона: `<start>.<end>[:<start>.<end>]`. Например:

  ```
  -bmyarea:0x1000.0x2000:0x3000.0x4000
  ```

  определяет, что `myarea` располагается от `0x1000` до `0x2000` и от `0x3000` до `0x4000`.

- `-fmots19` – формат выходного файла Motorola S19.

- `-blit:<address ranges>` – назначение адресных интервалов для области `lit`. Формат: `<start address>[<.end address>]`, где `address` – байтовые адреса. Файл запуска написан так, что любая память, не используемая этой областью, будет использована областями, которые следуют за ней, так что это фактически объявляет размер FLASH памяти. Например, некоторые типичные значения:

  ```
  -blit:0x1000.0xFFDF     для MSP430F149
  -blit:0xF000.0xFFDF     для MSP430F122
  ```

- `-bdata:<address ranges>` – назначение адресных интервалов для области или секции `data`, являющиеся памятью данных MSP430. Например, типичные значения:

  ```
  -bdata:0x200.0x0A00     для MSP430F149
  -bdata:0x200.0x300      для MSP430F122
  ```

12. ИНСТРУМЕНТАРИЙ

12.1. Компрессор кода

Компрессор года (Code Compressor (tm)) является современным оптимизатором, который уменьшает заключительный размер вашей программы на 5%-18%. Он работает во всем объеме вашего кода, и ищет во всех файлах возможности уменьшить размер программы.

12.1.1. Преимущества

- Компрессор Кода уменьшает размер вашей программы. Он не мешает традиционной оптимизации и может уменьшать размер кода даже тогда, когда традиционные агрессивные методы оптимизации уже применялись.
- В отличие от других подобных схем, эта – первая из известных нам реализаций коммерческого компилятора для встроенных систем, который оптимизирует всю программу.
- Компрессор года не влияет на отладку на уровне исходного кода.

12.1.2. Недостаток

- Имеется небольшое увеличение времени выполнения из-за затрат на вызовы функций.

12.1.3. Требования совместимости

Чтобы сделать ваш код полностью совместимым с компрессором кода, обратите внимание, что косвенные вызовы функций должны быть выполнены через метку входа в функцию в области `func_lit`. См. 9.5. Области программы. Если вы используете Си, это выполняется автоматически. Так как MSP430 поддерживает косвенный вызов функции одной инструкцией (ценой увеличения времени выполнения), воздействие на размер кода является незначительным.

12.1.4. Временная дезактивация компрессора кода

Иногда желательно временно отключить компрессор кода. Например, если код чрезвычайно чувствителен ко времени выполнения, и вы не можете позволить затраты из-за потери машинных циклов в дополнительных вызовах и возвратах из функций. Вы можете сделать это фрагментами кода с парой команд заключенных в скобки:

```
asm(".nocc_start");
...
asm(".nocc_end");
```

Компрессор года игнорирует команды во фрагменте между этими директивами ассемблера.

Заголовочный файл `msp430def.h` содержит два определения для использования в Си программе:

```
COMPRESS_DISABLE; // disable Code Compressor
COMPRESS_REENABLE; // enable Code Compressor again
```

12.2. Система управления версиями

ПРОФЕССИОНАЛЬНАЯ версия программного обеспечения предоставляет набор инструментов управления конфигурацией и интерфейс среды разработки для управления вашим исходным кодом. Программное обеспечение командной строки GNU Revision Control System (RCS) – это утилиты Системы Управления Версиями (см. 1.13. Благодарности для замечаний по программному обеспечению GNU). RCS контролирует множественные версии исходных файлов, позволяя вам при необходимости просматривать их старейшие версии. Среда предоставляет простой интерфейс к RCS, который является достаточным для большей части общих задач. Чтобы решать более сложные задачи, вы должны использовать утилиты командной строки RCS непосредственно. Эта страница описывает некоторые из наиболее общих функций RCS (посетите http://www.gnu.org для получения полной документации по GNU RCS).

12.2.1. Репозиторий системы управления версиями

Под управлением RCS для каждого файла хранится главная запись, содержащая все изменения, сделанные в файле каждой версии. Обычно RCS репозиторий – подкаталог по имени RCS в месте расположения исходного файла. Среда разработки создает репозиторий автоматически.

Каждое изменение файла имеет номер изменения и опциональную метку. Вы ссылаетесь на нужную версию по номеру или метке. Метка полезна для сохранения снимка частного набора изменений (например, перед тем, как вы выпускаете ваше программный продукт).

При расширенном использовании, вы можете изменять даже старейшую версию и “объединять” в ваших изменениях, или иметь множество изменений для того же самого файла, сделанное различными людьми и согласовывать различные изменения (при отсутствии конфликтов). Тема расширенного использования в этом документе не обсуждается.

12.2.2. Добавление и изменение файлов репозитория

Чтобы добавить новую версию файла в репозиторий, вы используете команду checkin (утилита `ci`). Для изменения файла в репозитории, вы используете команду checkout (утилита `co`). В простейшем случае, специальная опция к `ci` проверяет файл и затем выполняет непосредственно checkout, чтобы вы могли продолжать модифицировать файл.

Среда разработки использует `ICCV7 for 430` как имя регистрации файлов в репозитории.

12.3. Синтаксис ассемблера

Ассемблер использует описанный ниже синтаксис. Заметьте, что директивы нашего ассемблера отличаются от директив ассемблеров TI и IAR.

12.3.1. Имена

Все имена в ассемблере должны соответствовать следующей спецификации:

```
( '_' | [a-Z]) [[a-Z] | [0-9] | '_'] *
```

То есть имя должно начинаться или с подчеркивания (_) или с алфавитного символа, с последующими алфавитными символами, цифрами или подчеркиваниями. В этом документе, имена и символы являются синонимами. Имя является или именем символа, который является константой, или именем метки, которая является значением программного счетчика (PC) в данном месте. Имя может иметь длину до 30 символов. Имена чувствительны к регистру, исключая мнемоники команд и директивы ассемблера.

12.3.2. Видимость имен

Символ может использоваться или только внутри модуля программы, или может быть сделан видимым другим модулям. В первом случае символ называется **локальный**, во втором случае называется **глобальный**.

Если имя не определено внутри файла, в котором оно используется, то подразумевается, что оно определено в другом модуле, и его значение будет разрешено компоновщиком. Компоновщик иногда упоминается более точно как перемещающий компоновщик, потому что одна из его целей состоит в перемещении значений глобальных символов к их конечным адресам.

12.3.3. Числа

Если число имеет префикс 0x или $, оно считается шестнадцатеричным. Пример:

```
10
0x10
$10
0xBAD
0xBEEF
0xC0DE
-20
```

12.3.4. Формат входного файла

Входным файлом ассемблера должен быть ASCII файл, который соответствует некоторым соглашениям. Каждая строка должна иметь форму:

```
[label: [:]] [command] [operands] [;comments]
[] – опциональное поле
// Комментарии
```

Каждое поле должно отделяться от другого поля последовательностью из одного или большего числа “пробельных символов”, которые являются символами пробела или табуляции. Весь текст после спецификатора комментария (точка с запятой, или двойная наклонная черта вправо //) и до символа новой строки игнорируется. Входной файл имеет свободный формат. Например, вы не обязаны начинать метку в 1-й позиции строки.

12.3.5. Метки

Имя, сопровождаемое одним или двумя двоеточиями, означает метку. Значением метки является значение программного счетчика (PC) в данном месте программы. Метка с двумя двоеточиями является глобальным символом, то есть видимой другими модулями.

12.3.6. Команды

Командой может быть инструкция MSP430, директива ассемблера или макро вызов. Поле операндов содержит параметры команды. Здесь не описываются команды MSP430, т.к. используется стандарт мнемоник TI. Используйте документацию TI с описаниями команд.

12.3.7. Выражения

Операнд команды может включать выражение. Например, режим прямой адресации является простым выражением:

```
lds R10,asymbol
```

Выражение `asymbol` – пример самого простого выражения, которое является только именем метки или символом. Общее описание выражения:

```
expr: term | ( expr ) | unop expr | expr binop expr
term: . | name | #name
```

Точка "." является текущим программным счетчиком. Круглые скобки () обеспечивают группировку. Приоритетность операторов описывается ниже. Выражения не могут быть произвольной сложности из-за ограничений информации о перемещаемости, сообщаемой компоновщику. Основное правило состоит в том, что выражение может иметь только один перемещаемый символ. Например,

```
lds R10,foo+bar
```

Является недопустимым, если оба символа `foo` и `bar` являются внешними.

12.3.8. Операторы

Следующая таблица содержит список операторов и их приоритетность. Операторы с более высоким приоритетом выполняются первыми. С переместимым символом может использоваться только оператор сложения (как с внешним символом). Все остальные операторы должны применяться к константам или к символам, разрешимым ассемблером (как к символам, определенным в файле).

Оператор	Функция	Тип	Приоритет
*	умножение	бинарный	10
/	деление	бинарный	10
%	модуль	бинарный	10
<<	сдвиг влево	бинарный	5
>>	сдвиг вправо	бинарный	5
^	побитовое исключающее ИЛИ	бинарный	4
&	побитовое И	бинарный	4
|	побитовое ИЛИ	бинарный	4
-	отрицание	унарный	11
~	дополнение до 1	унарный	11
<	младший байт	унарный	11
>	старший байт	унарный	11

12.3.9. "Точка" или программный счетчик

Если в выражении появляется точка (`.`), то вместо точки используется текущее значение программного счетчика (PC).

12.4. Директивы ассемблера

Директивы ассемблера нечувствительны к регистру символов.

- `.area <name> [(attributes)]`

Определяет область памяти для загрузки следующего кода или данных. Компоновщик собирает вместе все области с одним именем и объединяет их последовательно или с перекрытием (оверлей), и располагает их по абсолютным или перемещаемым адресам, в зависимости от атрибутов области. Атрибутом перемещаемости может служить один из следующих:

```
abs   - абсолютная область, или
rel   - перемещаемая область
```

с последующим

```
con   - располагать последовательно, или
ovr   - накладываться, т.е. оверлей
```

Начальный адрес абсолютной области определяется в файле ассемблера непосредственно, в то время как начальный адрес переместимой области определяется как опция команды компоновщика. Для области с атрибутом `con`, компоновщик размещает одноименные области последовательно одна за другой. Для области с атрибутом `ovr`, для каждого файла, компоновщик начинает область с того же самого адреса. Следующий пример иллюстрирует сказанное:

```
file1.o:
    .area text    <- 10 bytes, area text_1
    .area data    <- 10 bytes
    .area text    <- 20 bytes, area text_2
file2.o:
    .area data    <- 20 bytes
    .area text    <- 40 bytes, area text_3
```

В этом примере, `text_1`, `text_2`, и так далее – имена областей, используемые в этом примере. Здесь они не представляют данные индивидуальные идентификаторы. Положим, что начальный адрес области `text` установлен в нуль. Тогда, если область `text` имеет атрибут `con`, `text_1` начинался бы в 0, `text_2` в 10, и `text_3` в 30. Если область `text` имеет атрибут `ovr`, то `text_1` и `text_2` снова имели бы адреса 0 и 10 соответственно, но `text_3`, так как он находится в другом файле, также имел бы начальный адрес 0. Все области с тем же самым именем должны иметь те же самые атрибуты, даже если они используются в разных модулях. Примеры полных перестановок всех приемлемых атрибутов:

```
.area foo(abs)
.area foo(abs,con)
.area foo(abs,ovr)
.area foo(rel)
.area foo(rel,con)
.area foo(rel,ovr)
```

- `.ascii "strings"`
- `.asciz "strings"`

Эти директивы используются для определения строк, заключенных в пару разделителей. Разделителем может быть любой символ, если начальный разделитель соответствует конечному разделителю.

Между разделителями допустимы любые печатные символы ASCII и следующие символы в стиле Си, каждый из которых начинается с наклонной черты влево (\):

```
\e          эскейп
\b          забой
\f          перевод страницы
\n          перевод строки
\r          возврат каретки
\t          табуляция
\<до 3-х 8-ричных цифр> символ с кодом равным 8-ричному числу
```

Директива `.asciz` добавляет символ NUL (`\0`) в конец последовательности символов. Допустимо включать `\0` внутрь строки. Примеры:

```
.asciz "Hello World\n"
.asciz "123\0456"
```

- `.byte <expr> [,<expr>]*`
- `.word <expr> [,<expr>]*`
- `.long <expr> [,<expr>]*`

Эти директивы определяют константы. Три директивы обозначают константу байта, константу слова (2 байта) и константу длинного слова (4 байта) соответственно. Константы слова и длинного слова выводятся в формате “little endian” – первым в памяти размещается младший байт. Этот формат используется микроконтроллерами MSP430. Обратите внимание, что `.long` может иметь только константы как значения операндов. Остальные могут содержать переместимые выражения. Пример:

```
.byte 1, 2, 3
.word label, foo
```

- `.blkb <value>`
- `.blkw <value>`
- `.blkl <value>`

Эти директивы резервируют пространство памяти без присвоения значений. Число зарезервированных элементов задается операндом.

- `.define <symbol> <value>`

Определяет текстовую подстановку имени регистра. Всякий раз, когда символ используется внутри выражения и ожидается имя регистра, он будет заменен выражением `value`. Например:

```
.define quot R15
mov quot,R16
```

- **`.else`**

Формирует условное выражение вместе с предшествующим `.if` и последующим `.endif`. Если условное выражение `.if` истинно, то все операторы ассемблера от `.else` до завершающего `.endif` игнорируются. Иначе, если условное выражение `.if` ложно, то блок выражения `.if` игнорируется, а блок выражения `.else` будет обработан ассемблером. См. `.if`.

- **`.endif`**

Заканчивает условное выражение. См. `.if` и `.else`.

- **`.endmacro`**

Заканчивает макро макроопределение. См. `.macro`.

- `<symbol> = <value>`

Определяет числовую константу для символа `symbol`. Пример:

```
foo = 5
```

- `.if <symbol name>`

  Если `symbol name` имеет ненулевое значение, то следующий код до выражения `.else` или до выражения `.endif` (тот, кто встретится первым) будет ассемблирован. Условные выражения могут иметь вложенность до 10 уровней. Например:

  ```
  .if cond
  lds R10,a
  .else
  lds R10,b
  .endif
  ```

  Значение из `a` загрузится в `R10`, если символ `cond` не равен нулю, а значение из `b` загрузится в `R10`, если `cond` – нуль.

- `.include "<filename>"`

  Обрабатывает содержимое файла `filename`. Если файл не существует, то ассемблер пробует открыть файл с именем, созданным объединением пути, определенного ключом командной строки `-I`, со специфицированным именем файла. Пример:

  ```
  .include "registers.h"
  ```

- `.macro <macroname>`

  Определяет макрокоманду. Тело макрокоманды состоит из всех операторов от `.macro` до `.endmacro`. В теле макрокоманды допускается любой оператор ассемблера кроме другой макрокоманды. Внутри тела макрокоманды, выражение `@digit`, где `digit` – в диапазоне от 0 до 9, заменяется соответствующим параметром макрокоманды при ее вызове. Вы не можете определять имя макрокоманды, которое находится в противоречии с мнемоникой команды или директивой ассемблера. См. `.endmacro` и Макро Вызов. Например, следующее определяет макрокоманду `foo`:

  ```
  .macro foo
  lds @0,a
  mov @1,@0
  .endmacro
  ```

  Вызов `foo` с двумя параметрами:

  ```
  foo R10,R11
  ```

  эквивалентен следующему:

  ```
  lds R10,a
  mov R11,R10
  ```

- `.org <value>`

  Устанавливает программный счетчик (PC), значением `value`. Эта директива допустима только для областей с атрибутом `abs`. Обратите внимание, что `value` – адрес байта. Пример:

  ```
  .area interrupt_vectors(abs)
  .org 0xFFD0
  .dc.w reset
  ```

- `.globl <symbol> [, <symbol>]*`

  Делает символы, определенные в текущем модуле, видимыми в других модулях. Аналогично имени метки с двумя последующими двоеточиями (`::`). Иначе, символы являются локальными в текущем модуле.

- `<macro> [<arg0> [,<args>]*]`

  Вызов макрокоманды по имени как команды ассемблера, сопровождаемой рядом параметров. Ассемблер заменяет вызов макрокоманды ее телом, расширяя выражение в форме `@digit` соответствующим параметром макрокоманды. Вы можете определить большее количество параметров, чем необходимо макрокоманде, но будет ошибкой, если вы определите меньшее количество параметров, чем необходимо Пример:

  ```
  foo bar,x
  ```

  Вызывает макрокоманду по имени `foo` с двумя параметрами: `bar` и `x`.

12.5. Команды ассемблера

Эта глава перечисляет все поддерживаемые TI MSP430 команды. За полной информацией о системе команд обращайтесь к документации TI. Имеется всего 27 команд MSP430. В таблицах `{.siz}` означает опциональный суффикс `.w` или `.b`. По умолчанию принимается `.w`. Суффикс `.w` означает 2-байтный операнд, суффикс `.b` означает 1-байтный операнд.

Таблица 1: Основные команды

Пересылка данных и арифметика		
mov{.siz} src,dst	add{.siz} src,dst	addc{.siz} src,dst
sub{.siz} src,dst	subc{.siz} src,dst	dadd{.siz} src,dst
and{.siz} src,dst	bic{.siz} src,dst	bis{.siz} src,dst
xor{.siz} src,dst		
Логика		
bit{.siz} src,dst	cmp{.siz} src,dst	
Передача управления		
call dst	reti	jmp label
jeq/jz label	jne/jnz label	jc label
jnc label	jn label	jge label
lj label		
Вращение		
rrc{.siz} dst	rra{.siz} dst	
Разное		
push src	swpb dst	sxt dst

Заметим - операнд `dst` не может быть непосредственным, даже в команде CMP.

Однако, используя семь режимов адресации и 16 регистров, некоторые из которых используются как быстрые генераторы констант, может быть создано много эмулированных команд. Ниже показаны обычно используемые эмулированные команды. Заметим, что возврат из функции `ret` является эмулированной командой (`mov @sp+,pc`)!

Таблица 2: Основные 'мулированные команды

Арифметика		
adc{.siz} dst	dadc{.siz} dst	dec{.siz} dst
decd{.siz} dst	inc{.siz} dst	incd{.siz} dst
sbc{.siz{} dst		
Логика		
inv{.siz} dst	rla{.siz} dst	rlc{.siz} dst
Данные		
clr{.siz} dst	clrc (очистка переноса)	clrn (очистка минуса)
clrz (очистка нуля)	pop dst	setc (установка переноса)
setn (установка минуса)	setz (установка нуля)	tst{.siz} dst
Управление		
br dst (длинный переход)	dint (запрет прерываний)	eint (разрешение прерываний)
nop	ret	

12.5.1. Режимы адресации

Имеется семь режимов адресации. Адресация может быть:

1. Регистровая – `Rn`
2. Индексная – `X(Rn)`, где X является 16-битным смещением
3. Символическая –`ADDR`, сокращение для `ADDR_offset(PC)`.
4. Абсолютная – `&ADDR`.
5. Косвенная регистровая – `@Rn`. Не может использовать операнд приемник.
6. Косвенная автоинкрементная – `@Rn+`. Не может использовать операнд приемник.
7. Непосредственная –`#N`, сокращение для `@PC+` с непосредственным значением в слове после команды. Не может использовать операнд приемник.

Регистры R2 и R3 также являются генераторами констант. Константы 0, + /-1, + 2, + 4 и + 8 могут быть сгенерированы, кодируя эти регистры с соответствующими режимами адресации. Это выполняется ассемблером автоматически.

12.6. Операции компоновщика

Основная цель компоновщика состоит в том, чтобы объединить множество объектных файлов в выходной файл, подходящий, для загрузки в программатор или симулятор. Компоновщик может также осуществлять ввод из “библиотеки”, которая в основном является файлом, содержащим множество объектных файлов. При создании выходного файла, компоновщик разрешает любые ссылки между входными файлами. С некоторыми подробностями, шаги компоновки включают:

1. Постановка файла запуска первым связываемым файлом. Файл запуска инициализирует среду исполнения программы Си.
2. Присоединение любых библиотек, которые вы явно запросили (или, как в большинстве случаев, которые запросила среда разработки) к списку файлов для связывания. Библиотечные модули, на которые имеются прямые или косвенные ссылки, будут участвовать в связывании. Все определенные пользователем объектные файлы (например, ваши программные файлы) будут связаны.
3. Добавление стандартной библиотеки Си `libc430.a` к концу списка файлов.
4. Просмотр объектных файлов для поиска неразрешенных ссылок. Компоновщик отмечает объектный файл (возможно в библиотеке) который удовлетворяет ссылки и добавляет к списку неразрешенных ссылок. Этот процесс повторяется до тех пор, пока не останется ожидающих обработки неразрешенных ссылок.
5. Объединение областей всех отмеченных объектных файлов в выходной файл и генерация файлов карты и листинга если необходимо.

Наконец, если это – УСОВЕРШЕНСТВОВАННАЯ или ПРОФЕССИОНАЛЬНАЯ версия и если включена опция оптимизации Code Compressor (tm), то вызывается 12.1. Компрессор кода.

12.6.1. Распределение памяти

Компоновщик объединяет области программы из входных объектных файлов и назначает им адреса памяти на основании диапазонов адресов, переданных из командной строки (см. 11.4.5. Параметры компоновщика). Эти параметры обычно передаются из среды разработки на основании выбранного устройства. То есть в обычном случае, вы не должны делать что-нибудь дополнительно, и среда с компилятором корректно распределят память.

Если вы используете `#pragma text / data / lit / abs_address` чтобы назначить ваши собственные области памяти, вы должны вручную гарантировать, что их адреса не накладываются на адреса, используемые компоновщиком. Наложение адресов может не вызвать генерацию ошибки компоновщиком. Вы должны всегда проверять файл карты `.mp` (используйте меню *View>Map File*) для поиска потенциальных проблем.

12.7. Отладочный формат ImageCraft

Отладочный файл ImageCraft (расширение .dbg) имеет частный ASCII формат, который описывает отладочную информацию. Компоновщик генерирует “стандартный” выходной отладочный формат непосредственно в дополнение к этому файлу. Например, AVR компилятор генерирует файл в формате COFF, который является совместимым с AVR Studio, а HC12 компилятор генерирует файл карты формата P$E. Документируя интерфейс отладки, мы надеемся, что отладчики могут выбрать использование этого формата.

Текущая версия этого интерфейса описана на нашем веб-сайте:

http://www.imagecraft.com/software/ImageCraft_debug_format.html

12.7.1. Отладчик NoICE

Предпочтительным отладчиком для ICCV7 for 430 является NoICE430 от производителя http://www.noicedebugger.com. По соглашению с авторами отладчика вы можете также приобрести NoICE на веб-сайте ImageCraft. NoICE является современным отладчиком с графическим интерфейсом пользователя на основе Windows, поддерживающий полную отладку на уровне исходного кода Си, за весьма привлекательную цену.

12.8. Библиотекарь

Библиотека – это коллекция объектных файлов в специальной форме, которую понимает компоновщик. Когда ваша программа непосредственно или косвенно ссылается на компонент объектного файла библиотеки, компоновщик извлекает из него библиотечный код и связывает его с вашей программой. Поддерживаемая стандартная библиотека – `libc430.a`, которая содержит стандартные функции Си и функции, специфичные для MSP430.

Однако бывают случаи, когда вам необходимо изменить существующую или создать новую библиотеку. Для этих целей предусмотрен инструмент командной строки `ilibw.exe`.

Заметьте, что библиотечный файл должен иметь расширение `.a`. См. 12.6. Операции компоновщика.

12.8.1. Компиляция файла в библиотечный модуль

Каждый библиотечный модуль является просто объектным файлом. Следовательно, чтобы создать библиотечный модуль, вы должны скомпилировать исходный файл в объектный. Для этого надо открыть файл в среде разработки, и вызвать команду *File>CompileFileToObject*.

12.8.2. Распечатка содержания библиотеки

В окне командной строки, смените каталог на тот, где находится библиотека, и дайте команду `ilibw -t <library>`. Например,

```
ilibw -t libc430.a
```

12.8.3. Добавление или замена модуля

Для добавления или замены модуля необходимо:

1. Компилировать исходный файл в объектный модуль.
2. Копировать библиотеку в рабочий каталог.
3. Использовать команду `ilibw -a <library> <module>` для добавления или замены модуля.

Например, следующие операции заменят функцию `putchar` в `libc430.a` вашей версией:

```
cd \iccv7430\libsrc.430
<modify putchar() in iochar.c>
<compile iochar.c into iochar.o>
copy \iccv7430\lib\libc430.a ; copy library
ilibw -a libc430.a iochar.o
copy libc430.a \iccv430\lib ; copy back
```

Команда `ilibw` создает библиотечный файл, если он не существует. Чтобы создать новую библиотеку, передайте `ilibw` имя нового библиотечного файла.

12.8.4. Удаление модуля

Командный ключ `-d` удаляет модуль из библиотеки. Например, следующие операции удаляют `iochar.o` из библиотеки `libc430.a`:

```
cd \iccv7430\libsrc.430
copy \iccv7430\lib\libc430.a ; copy library
ilibw -d libc430.a iochar.o ; delete
copy libc430.a \iccv7430\lib ; copy back
```