Компилятор ANSI C и среда разработки для Atmel AVR

Версия 7.XX

IMAGECRAFT C COMPILER FOR ATMEL AVR v.7.XX

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

СОДЕРЖАНИЕ

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 27 февраля 2006 г.

1.1.2. Торговые марки и авторские права

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 AVR 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 \iccv7avr\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:\iccv7avr\drivers`, заменив `c:\iccv7avr` 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 AVR 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:\iccv7avr\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:\iccv7avr\bin\davr.dll` в `c:\iccv7avr\bin\davr.dll.orig`.
2. Переименуйте `c:\iccv7avr\bin\davrnet.dll` в `c:\iccv7avr\bin\davr.dll`.
3. В директории Windows (например, `c:\windows` или `c:\winnt`), создайте файл по имени `iccavr.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:\iccv7avr\dongle_license`, вы должны написать:

   ```
   [dinkey]
   DinkeyNetPath=\\foo\c\iccv7avr\dongle_license
   ```

1. Когда вы запустите ICCV7 for AVR 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 AVR`.
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 AVR” 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 радаваемые допросы (FAQ) по продукту находятся по следующему адресу:

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

У нас имеется список рассылки, называемый icc-avr, предназначенный для пользователей нашего продукта ICCV7 for AVR. 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. Обновление 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. `libcavr.a` – базовая библиотека, содержащая стандартную библиотеку Си и специфичные для Atmel AVR 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.
- `.eep` – выходной Intel HEX файл, содержащий данные инициализации EEPROM.
- `.cof` – выходной отладочный файл в формате COFF.
- `.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>:<vector> <func2>:<vector> ...`

  Объявляет функции как обработчики прерываний, чтобы компилятор генерировал команды возврата из прерывания вместо команд обычного возврата и сохранял и восстанавливал все регистры, используемые функциями прерываний. Также генерируются векторы прерываний, основанные на их номерах. См. 8.13. Обработка прерываний. Эта прагма должна предшествовать определениям функций.

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

  Определяет, функции, которые не должны генерировать код, предохраняющий регистры `volatile`. Об использовании регистров см. 9.2. Интерфейс ассемблера и соглашения о вызовах. Эта прагма обычно используется в RTOS, где ядро системы управляет регистрами непосредственно. См. 8.11. Си-задачи.

- `#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. 9.7. 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. 8.7. 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
  ```

- Ассемблер Atmel использует адрес слова или байта в зависимости от инструкции. Ассемблер ICCV7 for AVR всегда использует адреса байта, если явно не используется оператор адресации слова. См. 12.3. Синтаксис ассемблера.
- Указатели на функции содержат дополнительный уровень косвенности из-за требования Компрессора Кода. См. 1.12.1. Компрессор кода – Code Compressor ™.

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. Некоторые из 16-разрядных процедур арифметики multiply/divide/modulo были написаны Atmel. Людям, помогавшим библиотечными арифметическими процедурами с плавающей точкой и длинными целыми, которым мы вечно благодарны: Jack Tidwell, Johannes Assenbaum, and Everett Greene. Утилита Make - Jacob Navia. Попробуйте недорогой Win32 компилятор Jacob Navia на http://www.cs.virginia.edu/~lcc-win32. Инструментарий также включает программу AvrCalc, написанную Jack Tidwell. Построитель приложений написан Andy Clark. Попробуйте его программу ACIDE и его AVR+GameBoy (tm) проект: http://pages.zoom.co.uk/andyc/. Код ISP написан Claudio Lanconelli. Попробуйте его Pony Programmer на http://www.lancOS.com. Frans Kievith переписал некоторые из библиотечных функций на ассемблере. David Raymond помог самыми малыми функциями деления, модуля, умножения. Заголовочные файлы io????v.h написаны Johannes Assenbaum.

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 под \iccv7avr\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 `\iccv7avr\examples.avr` 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 `\iccv7avr\examples.avr`. Этот файл создается в формате Intel HEX. Большинство программаторов и симуляторов Atmel AVR понимают этот формат, и вы можете загрузить эту программу в ваш контроллер. Это все, что нужно для создания данной программы.

Если вы хотите протестировать вашу программу в отладчике, который распознает отладочную информацию в формате COFF, например AVR Studio, то вы должны выбрать COFF как формат выходного файла в меню *Project>Options...>Compiler>Output Format*.

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

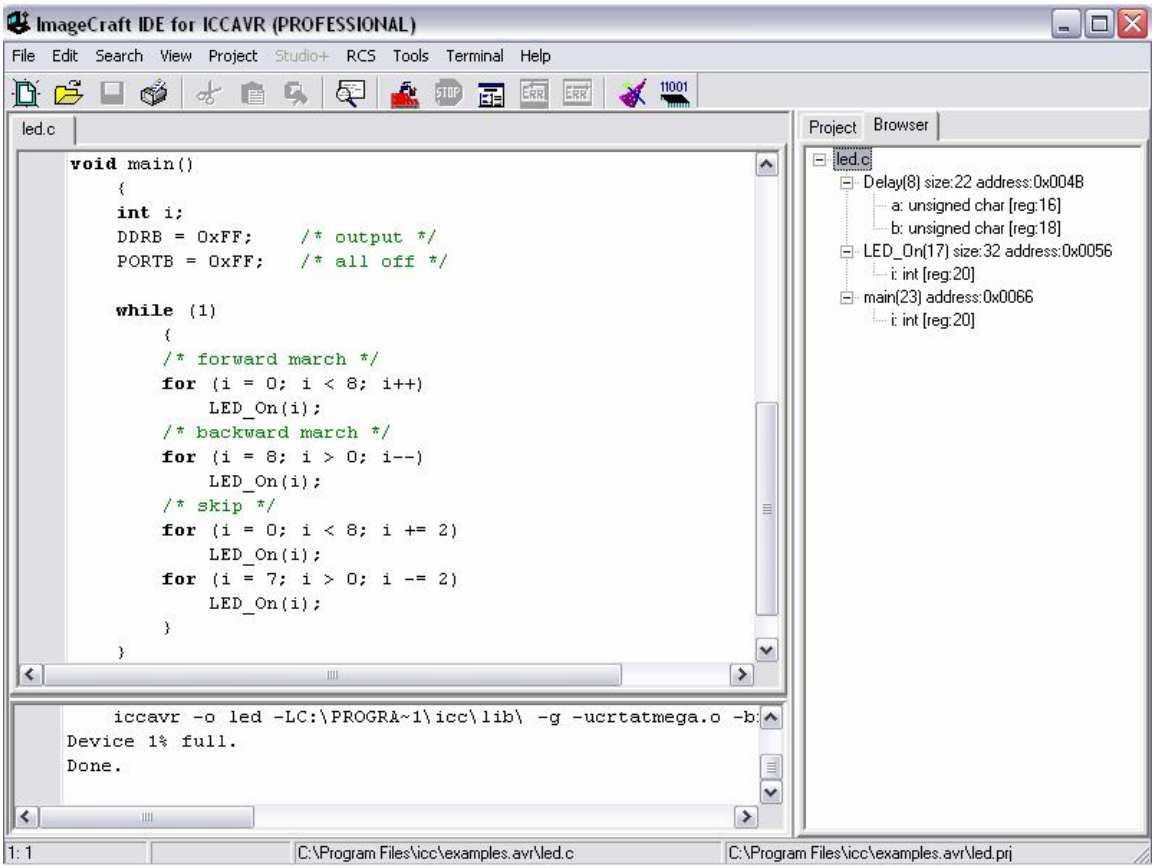
Программа на Си должна определять функцию с именем `main`. Компилятор связывает вашу программу со стартовым кодом и библиотечными функциями в "исполнимый" файл. Назначение стартового кода подробно описано в разделе 7.2. Файл запуска. Программа на Си нуждается в настройке целевой среды, и стартовый код выполняет эту функцию. В общем случае, программа `main` производит некоторую инициализацию и затем выполняет бесконечный цикл. Для примера, изучите файл `led.c` в каталоге `\iccv7avr\examples`:

```
#include <io8515v.h>

/* This seems to produce the right amount of delay
 * for the LED to be seen */

void Delay(){
unsigned char a, b;

    for (a = 1; a; a++)
        for (b = 1; b; b++);
}

void LED_On(int i){

    PORTB = ~BIT(i); // low output to turn LED on
    Delay();
}

void main(){
    int i;

    DDRB = 0xFF; /* output */
    PORTB = 0xFF; /* all off */
    while(1){
        /* forward march */
        for (i = 0; i < 8; i++)
            LED_On(i);
        /* backward march */
        for (i = 8; i > 0; i--)
            LED_On(i);
       /* skip */
       for (i = 0; i < 8; i += 2)
            LED_On(i);
       for (i = 7; i > 0; i -= 2)
            LED_On(i);
   }
}
```

Функция `main` очень проста. После инициализации регистров ввода/вывода, выполняется бесконечный цикл, изменяя порядок зажигающихся светодиодов в виде перемещающейся комбинации. Свечение светодиодов меняется в процедуре `LED_On`, которая просто записывает необходимые значения в порт вывода. Так как ЦПУ работает очень быстро, `LED_On` вызывает цикл задержки, чтобы рисунок свечения мог быть замечен. Так как точность задержки не существенна, пара вложенных циклов дает приблизительную подходящую величину. Если точность времени задержки важна, программа должна использовать регистры таймера для подсчета времени. Пример `8515intr.c` очень похож и показывает также насколько просто написать обработчик прерывания на Си.

2.3. Краткий обзор среды разработки

Главное окно среды разработки 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 4.18. 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 группы 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 `deficcavr.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 ошибок 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 типа AVR Studio. 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 AVR Studio. 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`. Обычно 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.

- **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 jкне 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 `\iccv7avr\bin\deficcavr.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 `\iccv7avr\include` (замените `\iccv7avr` 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 `\iccv7avr\lib` (замените `\iccv7avr` 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 `libcavr.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. Например, AVR Studio понимает формат COFF.
- **Optimizations** – управляет уровнем и типом оптимизации из следующих вариантов:
  - **Enable Code Compression** – только в УСОВЕРШЕНСТВОВАННОЙ и ПРОФЕССИОНАЛЬНОЙ версиях. Вызывает оптимизатор Code Compressor (tm) для устранения дублирующих фрагментов кода. Его операции прозрачны, но вы должны изучить описание на 12.1. Компрессор кода, чтобы ознакомиться с его ограничениями.
  - **Enable Global Optimizations** – только в ПРОФЕССИОНАЛЬНОЙ версии. Вызывает глобальный оптимизатор кода и быстродействия программ.
- **AVR Studio Version (COFF)** – определяет версию используемого AVR Studio. Studio 4.0 и выше позволяют исходным и COFF файлам быть в различных каталогах, а Studio 4.06 и выше могут раскрывать элементы структур (генерируется только ICCAVR PRO).
- **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"**.
- **Memory Sizes** – определяет объем памяти программ и данных в устройстве. Поддается изменению только, если вы выбираете "**Custom**" в списке выбора устройства. Память данных относится к внутренней SRAM.

  Эта опция также позволяет вам определять размер EEPROM. Обратите внимание, чтобы обойти аппаратную ошибку в AVR, когда вы имеете инициализированные данные в EEPROM, адрес 0 не используется.
- **Text Address** – обычно текст (код) начинается сразу после таблицы векторов прерываний. Например, код начинается в `0xD` (адрес слова) для 8515 и `0x30` для устройств Mega. Однако если вы не используете все прерывания, вы можете начинать ваш код раньше, если это не конфликтует с таблицей векторов. Изменяется только, если вы выбираете **"Custom"** в списке выбора устройств.
- **Data Address** – определяет начало памяти данных. Обычно это – `0x60`, сразу после регистров и портов ввода/вывода. Изменяется только, если вы выбираете **"Custom"** в списке выбора устройств. Игнорируется, если вы выбираете внешнюю память SRAM. Данные начинаются в начале внешней SRAM, если она выбрана.
- **Use Long JMP/CALL** – определяет поддержку устройством команд длинный `jmp` и длинный `call`.
- **Enhanced Core** – определяет поддержку устройством расширенных основных команд типа аппаратного умножения, `lpm z+, movw` и т.д.
- **IO Registers Offset Internal SRAM** – определяет, смещают ли регистры ввода/выывода начало внутренней SRAM или нет. Например, SRAM 8515 начинается в `0x60`, после пространства ввода/вывода и достигает 512 байт. Для устройства Mega603, регистры ввода/вывода перекрывают пространство SRAM, и, следовательно, SRAM начинается в 0 (но в непригодной для использования области до конца `0x60`) и достигает 4096 байт. Изменяется только, если вы выбираете **"Custom"** в списке выбора устройств.
- **Bootloader Options** – допускается только для устройств, которые поддерживают начальные загрузчики, такие, как новые устройства ATmega. Вы можете определять, содержит ли проект код приложения или код начального загрузчика, и насколько велик размер начального загрузчика. См. 8.12. Начальный загрузчик.
- **Use ELPM** – автоматически устанавливается, если вы выбираете “Bootloader Options” для устройства, имеющего больше, чем 64 КБ памяти FLASH. Это позволяет иметь корректный доступ к сегментам FLASH в загрузчике для этих устройств.
- **Internal vs. External SRAM** – определение типа данных SRAM вашей целевой системы. Если вы выбираете внешнюю SRAM, будут установлены правильные биты MCUCR.
- **PRINTF Version** – эта группа переключателей позволяет вам выбирать, с какой версией `printf` будет связана ваша программа. Больше возможностей потребует больший размер 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 ширине и полям точности, обеспечиваются `%ld, %lu, %lx`, и `%lX`.

- С плавающей запятой: добавляются форматы `%e`, `%f` и `%g` для плавающей запятой. Обратите внимание, что для не mega устройств, из-за большого объема кода, поддержка длинных не обеспечивается. Для mega устройств, все форматы и модификаторы доступны.

- **AVR Studio Simulator IO** – если флаг установлен, используйте библиотечные функции для интерфейса с окном терминала в AVR Studio (ТОЛЬКО V3.x). Обратите внимание, что вы должны скопировать файл `iostudio.s` в ваш исходный каталог. См. 10.2. Отладка COFF-кода и работа в AVR Studio.

- **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.

- **Strings in FLASH** – по умолчанию, символьные строки расположены во FLASH ROM и в SRAM, чтобы позволить простое смешивание указателей на строки и символы. (См. 8.3. Строки) Если вы хотите устранить перерасход памяти из-за размещения строк в SRAM, вы можете пометить эту опцию. Вы должны быть осторожными, выбирая правильные библиотечные функции. Например, для копирования символьной строки, расположенной этим способом, в буфер, вы должны использовать `cstrcpy()` функцию вместо стандартной функции `strcpy()`. (См. 7.8. Строковые функции)

- **Advanced** – эти опции дают вам более тонкий контроль над настройками компиляции:

  - **Return Stack Size** – компилятор использует два стека, один для адресов возврата из функций и обработчиков прерываний (аппаратный стек), и один для передачи параметров и локальных данных (программный стек). Эта опция позволяет вам управлять размером стека возврата. Размер программного стека не требует определения. См. 8.6. Стеки.

    Каждое обращение к функции или обработчику прерывания используют два байта стека возврата. Следовательно, вы должны оценить самый глубокий уровень ваших вызовов (то есть максимальное число вложенных процедур, которые ваша программа может вызвать и плюс прерывания) и ввести здесь соответствующий размер. **Программы, использующие плавающую точку или длинные целые, должны определять размер аппаратного стека, по меньшей мере, 30 байт.** Однако, в большинстве случаев, максимальный размер аппаратного стека в 50 байт должен быть достаточным. Так как аппаратный стек использует SRAM, если вы определите его слишком большим, стек может переполниться в глобальные переменные или в программный стек, вызвав сбои программы.

  - **Non Default Startup** – файл запуска всегда компонуется с вашей программой (см. 7.2. Файл запуска). В некоторых случаях, вы можете иметь различные файлы запуска, основанные на свойствах проекта. Эта опция позволяет вам определять имя файла запуска. Если имя файла – не полное имя пути, то файл запуска должен быть в одном из библиотечных каталогов.

  - **Unused ROM Fill Bytes** – заполняет неиспользуемые области ROM определенным целочисленным шаблоном.

  - **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*.

- **Do Not Use R20..R23** – заставляет компилятор не использовать R20 – R23 при генерации кода. См. 8.9. Глобальные регистры.

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.

4.15. 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 `\iccv7avr\bin\editors.ini`. Мы обеспечиваем информацию для некоторых из наиболее популярных редакторов в файле `\iccv7avr\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---”. Введите 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.

4.16. Опции внутрисхемного программатора

Инструментарий ISP поддерживает несколько адаптеров программаторов, включая STK-200/300 совместимый адаптер параллельного порта, адаптер параллельного порта DT-006 Dontronics, адаптер последовательный порта SI-Prog и интерфейс STK-500/AVRISP. Все, кроме интерфейса STK-500 управляется непосредственно средой ICCV7 for AVR IDE. STK-500 управляется, используя программу командной строки Atmel `stk500.exe`, которая является частью пакета Atmel AVR Studio (3.5X или выше).

4.16.1. Опции задержки

Для всех интерфейсов кроме STK-500, управляемых непосредственно средой, обычно не возникает каких-либо проблем с программированием контроллеров. Однако если вы с ними сталкиваетесь, вам может понадобиться скорректировать некоторые значения задержек синхронизации. Значения задаются в миллисекундах. Значения по умолчанию:

- Programming Delay – 20 миллисекунд
- TinyAVR and 90S1200 Delay – 15 миллисекунд
- Mega 103 Delay – 80 миллисекунд
- Mega 161 Delay – 60 миллисекунд
- Reset Delay – 20 миллисекунд (как минимум)

4.16.2. Опция пути STK-500

Для использования STK-500 или интерфейса Atmel AVR-ISP (который является только программирующей частью платы STK-500), вы должны выбрать полный путь к файлу `stk500.exe`, установленному Atmel AVR Studio. Studio 3.5X и Studio 4.X используют для этой программы различные каталоги по умолчанию; выберите версию, которую вы хотите использовать.

4.17. 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.

4.18. 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.

4.18.1. 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".

4.18.2. Контекстная подсветка

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

4.18.3. Назначения клавиш

Страница позволяет вам изменять назначения клавиш для команд редактирования.

4.18.4. 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.

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

а). 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 оно 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 ввод/вывод, управляемый прерыванием (пример этого доступен в каталоге `c:\iccv7avr\examples.avr`) или выводить текст 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 `libcavr.a` с вашей программой. В зависимости от целевого контроллера файл запуска может быть одним из следующих:

- `crtavr.o` – нормальный файл запуска.
- `crtatmega.o` – файл запуска ATmega. Использует `jmp __start` как вектор сброса.
- `crtavrram.o` – нормальный файл запуска с инициализацией внешней SRAM.
- `crtatmegaram.o` – файл запуска ATmega с инициализацией внешней SRAM.
- `crtboot.o` – файл запуска bootloader. Отличается от `crtavr.o` тем, что перемещает векторы прерывания. См. 8.12.1. Автономный загрузчик
- `crtboothi.o` – то же самое, что `crtboot.o`, но использует ELPM и инициализирует RAMPZ значением 1.

Вы можете создать ваш собственный файл запуска. Подробности см. в *Project>Options...>Target>Non-default Startup*. Файл запуска определяет глобальный символ `__start`, являющийся стартовой точкой вашей программы.

Функции файла запуска:

1. Инициализировать указатели программного и аппаратного стека.
2. Копировать инициализированные данные из области `idata` в области 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, с векторами, отличными от вектора сброса, изменять файл запуска не требуется. Компиляция или ассемблирование файла запуска требует специального ключа ассемблера (`-n`). Вы можете использовать IDE для компиляции файла запуска; используйте команду *File>Compile File...>Startup File To Object*.

Модификация и использование нового файла запуска:

```
cd \iccv7avr\libsrc.avr ; или место, где инсталлирован компилятор
<редактировать crtavr.s>
<открыть crtavr.s используя IDE>
```

выбрать *Compile File>To Object*>; генерация нового `crtavr.o`

```
copy crtavr.o ..\lib ; копирование в каталог библиотеки
```

7.3. Общее описание библиотеки Си

7.3.1. Исходный код библиотеки

Исходный текст библиотеки (`c:\iccv7avr\libsrc.avr\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 \iccv7avr\libsrc
unzip -s libsrc.zip
; unzip запросит пароль
```

7.3.2. Функции, специфичные для AVR

ICCV7 for AVR поставляется с набором функций 8.14. Доступ к UART, EEPROM, SPI, и другой периферии. Функции раздела 8.6.1. Проверка стека полезны для обнаружения его переполнения. Кроме того, наш веб-сайт содержит страницу с пользовательским исходным кодом.

Следующие файлы определяют регистры ввода/вывода, их биты и векторы прерывания устройств AVR:

```
io*v.h (io2313v.h, io8515v.h, iom128v.h, ... etc.)
```

Файл `macros.h` содержит полезные макрокоманды 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. 9.3. Функции, возвращающие нецелые значения и 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:\iccv7avr\examples.avr\` 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. Вы выбираете версию для использования в диалоговом окне “Compiler Options”.

  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 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 и 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.

7.8.1. Функции поддержки const char *

Эти функции выполняют ту же самую обработку, что и их дубликаты без префикса 'c' за исключением того, что они оперируют с постоянными строками в памяти FLASH.

- `void *cmemchr(const void *s, int c, size_t n);`
- `int cmemcmp(const char *s1, char *s2, size_t n);`
- `void *cmemcpy(void *dst, const void *src, size_t n);`
- `char *cstrcat(char *s1, const char *s2);`
- `int cstrcmp(const char *s1, char *s2);`
- `size_t cstrcspn(char *s1, const char *cs);`
- `size_t cstrlen(const char *s);`
- `char *cstrncat(char *s1, const char *s2, size_t n);`
- `int cstrncmp(const char *s1, char *s2, int n);`
- `char *cstrcpy(char *dst, const char *src);`
- `char *cstrpbrk(char *s1, const char *cs);`
- `size_t cstrspn(char *s1, const char *cs);`
- `char *cstrstr(char *s1, const char *s2);`

Следующие функции точно подобны соответствующим функциям без суффикса `x`, за исключением того, что они используют `elpm` вместо команды `lpm`. Это полезно для приложений начального загрузчика или если вы хотите поместить ваш код в верхние 64 КБ:

- `cmemcpyx, cmemchrx, cmemcmpx, cstrcatx, cstrncatx, cstrcmpx, cstrncmpx, cstrcpyx, cstrncpyx, cstrcspnx, cstrlenx, cstrspnx, cstrstrx, cstrpbrkx`

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:

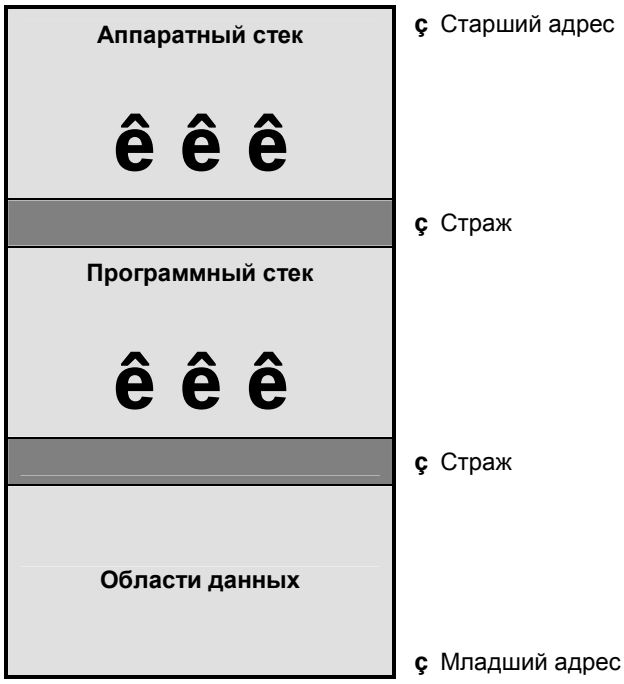
```
#include <stdarg.h>

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

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

7.10. Функции проверки стека

Несколько библиотечных функций предусмотрены для проверки переполнения стека. Рассмотрим следующую карту памяти. Если аппаратный стек врастет в программный стек, содержимое программного стека изменится. Это изменит значение локальных переменных и других элементов, помещенных в стек. Так как аппаратный стек используется для адресов возврата из функций, это может произойти, если ваша функция использует слишком глубоко вложенные вызовы.

Аналогично, переполнение программного стека в область данных изменило бы глобальные переменные или другие статически распределенные элементы (или кучу, если вы используете динамическую память). Это может случиться, если вы объявляете слишком много локальных переменных или если локальная составная переменная слишком большая.

Если вы часто используете функцию `printf`, строки формата могут занимать много места в области данных. Это также может способствовать переполнению стека. См. 8.3. Строки.



7.10.1. Резюме

Для использования функций проверки стека:

1. Использовать `#include <macros.h>`
2. Вставить `_StackCheck();` в вашем коде там, где вы хотите проверять стеки на переполнение. Это может быть в любом месте вашего кода, например, в вашей функции Watchdog Timer.
3. Когда `_StackCheck()` обнаруживает переполнение стека, вызывается функция `_StackOverflowed()` с целочисленным параметром 1, если переполнился аппаратный стек, и параметром 0, если переполнился программный стек.

4. По умолчанию библиотечная функция `_StackOverflowed()` переходит к адресу `0` и следовательно к перезапуску программы. Чтобы изменить это заданное по умолчанию поведение, напишите вашу собственную функцию `_StackOverflowed` в вашем исходном тексте. Это отменит поведение по умолчанию. Для отладки программы, ваша функция `_StackOverflowed` должна кое-что делать, чтобы указать катастрофическое условие, возможно, мигая светодиодом. Если вы используете отладчик, вы можете установить контрольную точку в функции `_StackOverflowed`, чтобы увидеть, когда она будет вызвана.

Прототипы этих двух функций существуют в заголовочном файле `macros.h`.

7.10.2. Стражи

Код запуска записывает байт стража по адресу сразу за областью данных и такой же байт по адресу за программным стеком. Если байты стражей изменятся – произошло переполнение стека.

Обратите внимание, что, если вы используете динамическое выделение памяти, вы должны пропустить байт стража в `_bss_end` для размещения вашей кучи. См. 7.7. Стандартная библиотека и функции памяти.

8. ПРОГРАММИРОВАНИЕ AVR

8.1. Доступ к специфическим ресурсам AVR

Сила Си в том, что, являясь языком высокого уровня, он позволяет вам обращаться к ресурсам низкого уровня целевых устройств. С такими способностями, имеются 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 `io*v.h` (`io8515v.h`, `iom103v.h`, и т.д.) определяют регистры ввода/вывода AVR, зависящие от типов устройств. Файл `macros.h` определяет много полезных макроопределений. Например, макроопределение UART_TRANSMIT_ON() может использоваться для включения UART (если существует в целевом устройстве).

Компилятор достаточно интеллектуален, чтобы генерировать одноцикловые команды, такие как `in`, `out`, `sbis`, и `sbi` при доступе к регистрам ввода/вывода. См. 8.8. Регистры ввода/вывода.

8.2. Данные программы и память констант

AVR – машина Гарвардской архитектуры, с памятью программ отделенной от памяти данных. Имеются несколько преимуществ такого решения. Часть из них в том, что отдельное адресное пространство позволяет устройству AVR обращаться к большему суммарному объему памяти, чем в стандартной архитектуре. В 8-разрядном ЦПУ с не Гарвардской архитектурой, максимальный объем памяти, который оно может адресовать, обычно составляет 64 КБ. Чтобы обращаться больше чем к 64 КБ, на таких устройствах обычно должна использоваться некоторая схема страничной памяти. С Гарвардской архитектурой, устройства Atmel AVR имеют несколько вариантов общего адресного пространства большего, чем 64 КБ, без использования страничной схемы.

К сожалению, Си был изобретен не на такой машине. Указатели Си являются или указателями на данные или указателями на функции, и уже правила Си определяют, что вы не можете полагать, что указатели на данные и функции могут быть преобразованы друг в друга. С другой стороны, с машиной Гарвардской архитектуры подобной AVR, даже указатель данных может указывать или на память данных или на программную память.

Не имеется никаких стандартных правил как это обрабатывать. Компилятор ImageCraft AVR использует квалификатор `const`, для указания, что элемент находится в памяти программ. Обратите внимание, что в объявлении указателя, спецификатор `const` может появляться в разных местах, в зависимости от того, квалифицирует ли он непосредственно переменную указатель или элемент, на который он указывает. Например:

```
const int table[] = { 1, 2, 3 };
const char *ptr1;
char * const ptr2;
const char * const ptr3;
```

Здесь `table` это таблица, расположенная в памяти программ, `ptr1` – элемент в памяти данных, который указывает на данные в памяти программ, `ptr2` – элемент в памяти программ, который указывает на данные в памяти данных, а `ptr3` – элемент в памяти программ, который указывает на данные в памяти программ. В большинстве случаев, элементы типа `table` и `ptr1` наиболее типичны. Компилятор Си генерирует команду LPM, чтобы обратиться к памяти программ.

Стандарт Си не требует, чтобы данные `const` были помещены в постоянную память, и в стандартной архитектуре это бы не имело значения, если бы не права доступа. Данное использование спецификатора `const` нестандартно, но это находится в пределах допустимых параметров стандарта Си. Обратите внимание, однако, что это порождает конфликт с некоторыми из стандартных определений функций Си.

Например, стандартный прототип функции `strcpy(char *dst, const char *src)` со спецификатором `const` второго параметра, означающим, что функция не изменяет параметр. Однако, в ICCV7 for AVR, спецификатор `const` показал бы, что второй параметр указывает на память программ. Наиболее вероятно, что это не тот случай, и таким образом, эти функции определены без квалификаторов `const`.

Наконец, обратите внимание, что только переменные `const` с классом хранения файл будут помещены во FLASH. Например, переменные, которые определены вне тела функции или имеют статический класс хранения, имеют класс хранения файл. Если вы объявляете локальные переменные с квалификатором `const`, они не будут помещены во FLASH, и результатом может быть непредсказуемое поведение. Компилятор выдает предупреждение, когда обнаруживает эту ситуацию.

8.2.1. ELPM и RAMPZ

Для доступа к FLASH памяти выше 64 КБ (как в M103 или M128), должна использоваться команда ELPM вместе с установкой бита RAMPZ в 1. Флаг (“Use ELPM”) был добавлен для этой цели на страницу 4.13. Опции компилятора: Целевой контроллер. Это полезно для начального загрузчика, который находится в верхней памяти. К сожалению, компилятор вообще не может определять, ссылается ли указатель FLASH памяти на верхнюю или нижнюю память. Вы должны устанавливать этот флаг вручную, если помещаете постоянные данные в верхней памяти, и вы обязаны установить бит RAMPZ самостоятельно.

8.2.2. Таблицы констант

Компоновщик не позволяет вам располагать один объект, пересекающий 64 КБ границу в Mega103 или Mega128. Для доступа к такому объекту потребовалось бы переключать бит RAMPZ при пересечении 64 КБ границы, а цена реализации такого гуляющего указателя настолько высока, что лишает его всякого смысла. Программисту лучше ограничиться размещением объекта целиком или ниже или выше 64 КБ границы и при доступе соответственно устанавливать RAMPZ.

8.3. Строки

Как объяснялось в разделе 8.2. Данные программы и память констант, разделение памяти программы и данных в Гарвардской архитектуре AVR создает некоторую сложность. Этот раздел объясняет эту сложность относительно символьных строк.

8.3.1. Строки

Компилятор помещает таблицы переключателей и элементы, объявленные как `const` в память программы. Последний непростой вопрос – размещение символьных строк. Проблема состоит в том, что в Си строки преобразованы в указатели на `char`. Если строки расположены в памяти программ, то либо все строковые библиотечные функции должны быть дублированы, чтобы обработать разные виды указателей, либо строки должны быть размещены также и в памяти данных. Компилятор ImageCraft предлагает для решения этих вопросов две опции.

8.3.2. Размещение строк по умолчанию

По умолчанию строки размещаются и в памяти данных и в памяти программ. Все ссылки к строкам относятся к их копиям в памяти данных. Чтобы гарантировать правильность их значений, при старте строки копируются из памяти программ в память данных. Таким образом, необходима только одна копия строковых функций. Это аналогично применению инициализированных глобальных переменных.

Если вы желаете сэкономить память, вы можете расположить строки в памяти программ, используя массивы символов `const`. Например:

```
const char hello[] = "Hello World";
```

В этом примере, `hello` может использоваться во всех контекстах, где может использоваться символьная строка, исключая использование как параметров строковых функций стандартной библиотеки Си, как объяснено ранее.

Функция `printf` расширена символом управления форматом `%S` для вывода строк, размещенных только во FLASH. Кроме того, были добавлены новые строковые функции, для поддержки строк только во FLASH. (См. 7.8. Строковые функции).

8.3.3. Размещение всех символьных строк только во FLASH

Вы можете указать компилятору, чтобы он разместил все символьные строки только во FLASH, устанавливая флаг “Strings In FLASH Only” в 4.13. Опции компилятора: Целевой контроллер. Как и прежде, вы должны быть осторожны при вызове библиотечных функций. Когда эта опция выбрана, действительный тип символьной строки – `const char *`, и вы должны гарантировать, что функция принимает соответствующий тип параметра. Помимо нового формата `const char *`, относящегося к 7.8. Строковые функции, функции `cprintf` и `csprint` принимают `const char *` как тип строки форматирования. См. 7.6. Стандартные функции ввода/вывода.

8.4. Заголовочные файлы io????v.h

Схема именования регистров ввода/вывода, определение битов и векторы прерываний стандартизированы в данных заголовочных файлах. Заголовочные файлы `io????v.h` определяют символические имена регистров ввода/вывода и битов AVR, номера векторов прерываний и биты конфигурации/защиты, если они реализованы (только Mega AVR). Имена регистров ввода/вывода и битов определены так же, как в справочных материалах (data sheets) за некоторыми исключениями и расширениями (обратите внимание, что иногда информационные листки имеют орфографические ошибки и опечатки!):

- Биты SREG не определены (в Си не используются).
- UART-status флаг OR определен как OVR (старый стиль ICCAVR) и DOR (новый стиль mega AVR).
- 16-разрядные регистры в последовательных адресах (например, регистры Timer/Counter1 или Stack Pointer) определены как `int`, так и `char`. Для использования определения `int` просто пишите имя регистра без суффикса L/H, например ICR1. Если для 16 разрядного доступа используется регистр процессора TEMP, ICCAVR сам создает правильную последовательность кода.
- Номера векторов прерываний определены так же, как в таблицах справочных данных “Reset and Interrupt Vectors” (исходный столбец) с префиксом “iv_”. Используйте их с прагмой `interrupt_handler`, например:

  ```
  #pragma interrupt_handler timer0_handler: iv_TIMER0_OVF
  ```

  Подробности см. в 8.13. Обработка прерываний.

Добавлены некоторые двойные определения, чтобы преодолеть наиболее раздражающие различия синтаксиса AVR:

- UART_RX и UART_RXC (*)
- UART_DRE и UART_UDRE (*)
- UART_TX и UART_TXC (*)
- EE_RDY и EE_READY
- ANA_COMP и ANALOG_COMP
- TWI и TWSI
- SPM_RDY и SPM_READY

(*) ПРИМЕЧАНИЯ

- Если устройство имеет USART вместо UART, имена векторов прерываний записываются как “iv_USART_” вместо “iv_UART_” (например iv_USART_RXC).
- Если устройство имеет больше чем один U(S)ART, имена векторов прерываний включают номер U(S)ART (например, iv_UART0_RXC).
- Также в версии 6.24A введен набор заголовочных файлов ассемблера с макросом для объявления обработчиков прерываний. См. 8.13. Обработка прерываний.

8.5. 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.

8.5.1. 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))
```

8.5.2. 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 бит не определен языком Си, и обычно битовые поля генерируют не лучший код. Мы настоятельно рекомендуем использовать побитовые операторы вместо этого.

8.6. Стеки

Сгенерированный код использует два стека: аппаратный стек, который используется вызовами процедур и программами обработки прерывания, и программный стек для размещения кадров параметров, временных и локальных переменных. Хотя это может показаться громоздким, использование двух стеков вместо одного дает наиболее ясное использование данных RAM.

Так как аппаратный стек используется, прежде всего, чтобы сохранять адреса возврата функций, он обычно намного меньше чем программный стек. Вообще, если ваша программа не использует свои или библиотечные функции с интенсивными вызовами, типа `printf` с `%f` форматом, то 16 байтный стек по умолчанию должен работать удовлетворительно. В большинстве случаев, максимального значения в 40 байт для аппаратного стека достаточно, если ваша программа не имеет глубоко вложенных рекурсивных процедур.

Аппаратный стек расположен в верхней памяти данных, а программный стек расположен несколько ниже. Размер аппаратного стека и размер памяти данных управляется установками в окне 4.13. Опции компилятора: Целевой контроллер. Область данных расположена, начиная с 0x60 или 0x100, после области ввода/вывода. Это позволяет области данных и программному стеку расти навстречу друг другу.

Если вы выбираете устройство с 32 КБ или 64 КБ внешней SRAM, то стеки размещены наверху внутренней SRAM и растут вниз к нижним адресам памяти. См. 9.6. Использование памяти программ и данных.

8.6.1. Проверка стека

Частый источник непредвиденного отказа программы – стек, переполняющийся в другие области памяти данных. Любой из двух стеков может переполниться, и при переполнении стека могут произойти непредсказуемые явления. Вы можете использовать 7.10. Функции проверки стека, чтобы обнаружить ситуации переполнения.

8.7. 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 части 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 uc;

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

Любая переменная Си может быть вызвана этим способом, 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. Не имеется также никакого способа управлять распределением регистровых переменных, так что ваша встроенная команда может потерпеть неудачу. Например, использование команды `ldi` требует, чтобы регистр был одним из 16 верхних регистров, но не имеется никакого способа запросить это, используя встроенный ассемблерный код. Не имеется также никакого способа сослаться на старшую половину 16-разрядного регистра.

Встроенный ассемблер может использоваться внутри или вне функции Си. Компилятор выравнивает каждую строку встроенного ассемблера для удобочитаемости. В отличие от ассемблера AVR, ассемблер ImageCraft позволяет размещать метки в любом месте (не только в первом символе строки) так, чтобы вы могли создавать метки ассемблера в вашем встроенном ассемблерном коде. Вы можете получить предупреждение об операторах `asm`, которые появляются вне тела функции, и игнорировать эти предупреждения.

8.8. Регистры ввода/вывода

К регистрам ввода/вывода, включая регистр состояния SREG, можно обращаться двумя способами. Адреса ввода/вывода между 0x00 и 0x3F могут использоваться командами IN и OUT, чтобы читать и записывать в регистры ввода/вывода, а адреса памяти данных между 0x20 и 0x5F могут использоваться с нормальными командами обращения к памяти и соответствующими способами адресации данных. Оба метода доступны в Си:

- Адреса памяти данных. Прямой адрес может использоваться непосредственно, косвенно через указатель и через приведение типа. Например, адрес SREG в памяти данных равен 0x5F:

```
unsigned char c = *(volatile unsigned char *)0x5F;
// read SREG
*(volatile unsigned char *)0x5F |= 0x80;
// turn on the Global Interrupt bit
```

Обратите внимание, что память данных от 0 до 31 относится к регистрам ЦПУ! Должна соблюдаться крайняя осторожность, чтобы непреднамеренно не изменить регистры ЦПУ.

Это – **предпочтительный метод**, так как компилятор автоматически генерирует команды низкого уровня типа `in`, `out`, `sbrs`, и `sbrc` при доступе к памяти данных в области регистров Ввода-Вывода.

- Адреса ввода/вывода. Вы можете использовать встроенный ассемблер и макроопределение препроцессора, чтобы обратиться к адресам ввода/вывода:

```
register unsigned char uc;
asm("in %uc,$3F"); // read SREG
asm("out $3F,%uc"); // turn on the Global Interrupt bit
```

Этот способ не рекомендуется, поскольку встроенный ассемблерный код может препятствовать компилятору в выполнении некоторых способов оптимизации.

Обратите внимание: чтобы читать биты выводов порта ввода/вывода, вы должны обратиться к PINx вместо PORTx, например, PINA вместо PORTA. Пожалуйста, обратитесь к документации Atmel за подробностями.

8.9. Глобальные регистры

Иногда в программе более эффективно использовать доступ к глобальным регистрам. Например, в обработчиках прерывания, вы можете захотеть увеличивать глобальную переменную, к которой должна обращаться другая часть программы. Использование регулярных глобальных переменных Си этим способом может потребовать больше накладных расходов чем желается в обработчиках прерываний из-за необходимости сохранения и восстановления регистров и затрат на доступ к памяти, где находятся глобальные переменные.

Вы можете заставить компилятор не использовать регистры R20, R21, R22, и R23, установив опцию *Compiler>Options...>Target>Do Not Use R20..R23*. В общем случае, вы не обязаны устанавливать эту опцию, так как компилятор может генерировать большую программу, используя меньшее число регистров, чем имеется. Вы не можете резервировать другие регистры кроме этих.

В редких случаях, когда ваша программа содержит сложные выражения, использующие длинные целые и числа с плавающей точкой, компилятор может пожаловаться, что не может скомпилировать такие выражения с данной выбранной опцией. В таких случаях вы должны будете упростить эти выражения.

Вы можете обращаться к этим регистрам в вашей программе Си, используя прагму:

```
#pragma global_register <name>:<reg#> <name>:<reg#>...
```

Например:

```
#pragma global_register timer_16:20 timer_8:22 timer2_8:23
extern unsigned int timer_16;
char timer_8, timer2_8;
..
#pragma interrupt_handler timer0:8 timer1:7
void timer0(){
    timer_8++;
}
```

Обратите внимание, что вы должны объявлять типы глобальных регистровых переменных. Они должны иметь типы `char`, `short` или `int`, и вы должны гарантировать правильную нумерацию регистров. 2-х байтный глобальный регистр использует номер регистра, который вы определили и номер следующего регистра для хранения своего содержимого. Например, “timer_16”, описанный выше имеет тип `unsigned int`, и занимает регистры R20 и R21.

Так как эти регистры находятся в наборе верхних 16-ти регистров AVR, для них будет сгенерирован очень эффективный код при назначении констант и т.п.

Библиотеки обеспечены версиями, которые компилируются с этой опцией и без нее, и среда автоматически выбирает правильную библиотечную версию, основанную на опциях проекта.

8.10. Абсолютная адресация памяти

Ваша программа может потребовать адресации абсолютных областей памяти. Например, внешние периферийные устройства обычно отображаются на особые адреса памяти. Это могут быть интерфейс с LCD или двухпортовой SRAM. Вы можете также располагать данные в специфическом месте для связи между начальным загрузчиком и приложением или между двумя отдельными процессорами, обращающимися к двухпортовой RAM.

В следующих примерах, предполагается, что имеются двухбайтный регистр управления LCD по адресу 0x1000, двухбайтовый регистр данных LCD по адресу (0x1002), а также 100-байтная двухпортовая SRAM, размещенная по адресу 0x2000.

8.10.1. Использование #pragma abs_address

В файле Си, поместите следующее:

```
#pragma abs_address:0x1000
unsigned LCD_control_register;
#pragma end_abs_address

#pragma abs_address:0x2000
unsigned char dual_port_SRAM[100];
#pragma end_abs_address
```

Эти переменные могут быть объявлены как “extern” по обычным правилам Си в других файлах. Обратите внимание, что вы не можете инициализировать их в объявлениях.

8.10.2. 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.

8.10.3. Использование встроенного ассемблера

Встроенный ассемблер использует синтаксис регулярного ассемблера, но его команды заключены в псевдофункцию `asm()`. В файле Си, предыдущий 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, что находится внутри операторов `asm`.

8.11. 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 обстоятельствах, это поведение может быть нежелательно. Например, если Вы используете RTOS (Real Time Operating System), RTOS 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.

8.12. Начальный загрузчик

Некоторые из новейших mega-устройств поддерживают bootloader – приложение начального загрузчика. Вы можете создавать загрузчик как автономное приложение или иметь одно приложение, которое содержит и основной код и код загрузчика.

8.12.1. Автономный загрузчик

Выберите желаемый размер начального загрузчика в *Project>Options...>Target*. Среда сделает следующее для автономного загрузчика, генерируя соответствующие флаги компилятора:

1. Начальный адрес программы перемещается в начало области начального загрузчика, оставляя пространство для векторов прерывания.
2. Если выходное устройство имеет меньше чем 64 КБ FLASH, используется 7.2. Файл запуска `crtboot.o`, иначе будет использован файл `crtboothi.o`. Файл `crtboot.o` отличается от стандартного файла запуска тем, что разрешает перемещение векторов в область загрузчика, изменяя регистр IVSEL. Файл `crtboothi.o` также устанавливает RAMPZ в 1 и использует инструкцию ELPM.
3. Если целевое устройство имеет больше чем 64 КБ FLASH, автоматически устанавливается переключатель “Use ELPM” и программа компонуется с модулем `libcavrhi.a`. Этот режим отличается от `libcavr.a` тем, что компилируется с ключом `-use_elpm`.
4. Среда генерирует для компоновщика ключ “`-bvector:0x????`”, чтобы переместить абсолютную область векторов в верхнюю память. Это позволяет той же самой прагме обработчика прерывания использоваться в вашем исходном тексте, является ли он нормальным приложением или начальным загрузчиком.

8.12.2. Комбинация основной программы и загрузчика

Если вы хотите поместить некоторые коды функций в отдельную область загрузчика, вы можете использовать расширение “`#pragma text:myarea`”, чтобы расположить функцию в вашей собственной области. Затем введите в панели редактирования *Project>Options...>Target>Additional Options*:

```
-bmyarea:0x1FC00.0x20000
```

Замените “`myarea`” на любое имя, а адресный интервал значениями по вашему выбору. Обратите внимание, что адресный интервал выражается в байтах. Вы также будете должны управлять любыми векторами прерываний загрузчика самостоятельно.

Помните, что, если вы создаете комбинацию главного и загрузочного приложения, обе части совместно используют ту же самую копию кода библиотеки Си, по умолчанию постоянно располагающейся в основной области приложения. Следовательно, согласно этой схеме, загрузчик не может стирать основное приложение, поскольку ему может потребоваться обращение к коду библиотеки Си.

8.13. Обработка прерываний

8.13.1. Си обработчики прерываний

Программы обработки прерываний могут быть написаны на Си. В файле перед определением функции вы должны сообщить компилятору, что функция является программой обработки прерывания, используя прагму:

```
#pragma interrupt_handler <func name>:<vector number>
```

“vector number” – номер прерывания. Обратите внимание, что vector number начинается с 1 и является вектором сброса. 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 векторы прерываний, основанные на vector number и архитектуре целевого контроллера.

Например:

```
#pragma interrupt_handler timer_handler:4
...
void timer_handler(){
...
}
```

Компилятор генерирует команду

```
rjmp _timer_handler ; для классических AVR
jmp  _timer_handler ; для устройств Mega AVR
```

по адресу 0x06 (адрес байта) для классических устройств, и по адресу 0x0C (адрес байта) для устройств Mega. Устройства Mega используют вектор прерывания длиной в 2 слова, по сравнению с 1 словом в классическом не Mega устройстве.

Вы можете помещать несколько имен в одну прагму interrupt_handler, разделяя их пробелами. Если вы желаете использовать одну программу обработки прерывания для нескольких векторов прерываний, объявите ее несколько раз с различными номерами векторов. Например:

```
#pragma interrupt_handler timer_ovf:7 timer_ovf:8
```

Заголовочные файлы Си ioXXXXv.h определяет непротиворечивые глобальные имена для номеров векторов прерываний, допуская полностью символическую прагму interrupt_handler и облегчая замену целевого устройства. Глобальные номера векторов прерываний именуются по шаблону “iv_<vector_name>” с <vector_name>, соответствующими справочным данным AVR. Например:

```
#pragma interrupt_handler timer0_handler: iv_TIMER0_OVF
#pragma interrupt_handler eep_handler: iv_EE_READY
#pragma interrupt_handler adc_handler: iv_ADC
```

Так как имена номеров прерываний определены в заголовочном файле, они могут быть легко изменены для другого устройства AVR включением другого заголовочного файла. Новые устройства должны, конечно, соответствовать аппаратным требованиям. Для имен, обеспечиваемых различными заголовками, см. файлы avr_c_lst и mega_c_lst в ICCV7 for AVR в каталоге include.

8.13.2. Ассемблерные обработчики прерываний

Вы можете написать обработчик прерывания на Ассемблере. Однако если вы вызываете функции Си внутри вашего ассемблерного обработчика, ассемблерная процедура должна сохранять и восстанавливать изменяемые 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).

Если вы используете ассемблерные процедуры обработки прерывания, вы должны определить векторы самостоятельно. Используйте атрибут “abs”, чтобы объявить абсолютную область (см. 12.4. Директивы ассемблера) и используйте директиву “.org”, чтобы назначить команду rjmp или jmp по правильному адресу. Обратите внимание, что директива “.org” использует байтовую адресацию.

```
; для всех устройств кроме ATMega
.area vector(abs) ; interrupt vectors
.org 0x6
rjmp _timer

; для устройств ATMega
.area vector(abs) ; interrupt vectors
.org 0xC
jmp _timer
```

Заголовочные файлы ассемблера “aioXXXX.s” поддерживают макрос для символического определения вектора прерывания. Синтаксис:

```
set_vector_<vector_name> <jumpto_label>
```

с <vector_name> как в справочных данных AVR и <jumpto_label> равным имени пользовательского обработчика. Примеры:

```
set_vector_TIMER0_OVF t0_asm_handler
set_vector_ADC adc_asm_handler
set_vector_UART0_DRE u0dre_asm_handler
```

В зависимости от типа контроллера макрорасширение может приводить к различному коду. Для имен, обеспечиваемых различными заголовками, см. avr_asm_lst, и файлы mega_asm_lst в каталоге include ICCAVR.

8.14. Доступ к UART, EEPROM, SPI, и другой периферии

Application Builder генерирует код инициализации, используя интерфейс мыши, облегчая использование встроенной периферии AVR. Кроме того, исходный текст к некоторым функциям высокого уровня находится в каталоге EXAMPLE (`c:\iccv7avr\examples.avr`, где `c:\iccv7avr` – каталог вашей установки). Чтобы использовать их, скопируйте исходные файлы в ваш рабочий каталог и сделайте любые необходимые модификации. Многие из этих файлов написаны пользователями ImageCraft. Может иметься больше примеров, чем перечислено здесь. Пожалуйста, ищите детали в каталоге.

8.14.1. UART

Заданные по умолчанию библиотечные функции `getchar` и `putchar` читают и записывают в UART, используя режим опроса. В каталоге `\icc\examples.avr` прилагается набор управляемых прерываниями буферизированных процедур, которые вы можете использовать вместо заданных по умолчанию процедур.

8.14.2. EEPROM

Доступен набор макросов и функций доступа к EEPROM. См. 8.15. Доступ к EEPROM. Функция нормального доступа включена в библиотеку, а функции доступа в реальном масштабе времени находятся в файлах `rteeprom.h` и `rteeprom.c` в каталоге EXAMPLES.

8.14.3. SPI

Файлы `spi.c` и `spi.h` в каталоге EXAMPLES содержат исходный текст примера. Используйте Application Builder, чтобы генерировать функцию инициализации SPI.

8.14.4. LCD

Файл `lcd.zip` в каталоге EXAMPLES содержит исходный текст и демонстрационную программу для стандартных Hitachi или Toshiba совместимых текстовых контроллеров LCD.

8.14.5. I2C

Файл `i2c.zip` в каталоге EXAMPLES содержит исходный текст и демонстрационную программу для запуска I2C в мастер-режиме.

8.15. Доступ к EEPROM

К EEPROM можно обращаться во время выполнения программы, используя библиотечные функции. Используйте `#include <eeprom.h>` перед вызовом этих функций:

- **`EEPROM_READ`**`(int location, object)`

  Этот макрос вызывает функцию **`EEPROMReadBytes`** (см. ниже), чтобы прочитать `object` данных из `location(s)` EEPROM. "`object`" может быть любой переменной программы, включая структуры и массивы. Например,

```
int i;
EEPROM_READ(0x1, i); // читать 2 байта в i
```

- **`EEPROM_WRITE`**`(int location, object)`

  Этот макрос вызывает функцию **`EEPROMWriteBytes`** (см. следующий раздел) чтобы записать `object` данных в `location(s)` EEPROM. "`object`" может быть любой переменной программы, включая структуры и массивы. Например,

```
int i;
EEPROM_WRITE(0x1, i); // писать 2 байта в 0x1
```

Фактически имеются 3 набора макросов и функций:

- Большинство классических и mega AVR
- AVR с 256 байтами EEPROM
- MegaAVR с расширенным вводом/выводом

Среда предопределяет некоторые макросы (например, ATMega168), чтобы использовать правильные макросы и функции при включении заголовочного файла `eeprom.h`, для использования имен, данных здесь для этих макросов и функций.

8.15.1. Инициализация EEPROM

EEPROM может быть инициализирована в исходном файле программы размещением глобальной переменной в специальной области, называемой "`eeprom`". В исходном коде Си это может быть выполнено, используя прагмы. См. 9.7. Области программы для обсуждения различных областей программы. Результирующий файл – `<output file>.eep`. Например,

```
#pragma data:eeprom
int foo = 0x1234;
char table[] = { 0, 1, 2, 3, 4, 5 };
#pragma data:data
...
int i;
EEPROM_READ((int)&foo, i); // i now has 0x1234
```

Вторая прагма необходима, чтобы сбросить имя области данных обратно к заданному по умолчанию "`data`". Обратите внимание, чтобы обойти аппаратную ошибку в AVR, для инициализированных данных EEPROM адрес 0 не используется.

Обратите внимание, что при использовании внешнего объявления данных (например, `foo` в другом файле), вы не используете прагму. Например, в другом файле:

```
extern int foo;
int i;
EEPROM_READ((int)&foo, i);
```

8.15.2. Внутренние функции

Следующие функции могут использоваться непосредственно, если необходимо, но макрос, описанный выше должен удовлетворить большинству, если не всем ситуациям.

- `unsigned char` **`EEPROMread`**`(int location)`

  Читает байт из определенной области `location` EEPROM.

- `int` **`EEPROMwrite`**`(int location, unsigned char byte)`

  Записывает `byte` в определенную область `location` EEPROM. Возвращает `0` при удаче.

- `void` **`EEPROMReadBytes`**`(int location, void *ptr, int size)`

  Читает `size` байт, начинающихся в `location` EEPROM, в буфер, указанный `ptr`.

- `void` **`EEPROMWriteBytes`**`(int location, void *ptr, int size)`

  Записывает `size` байт содержимого буфера, указанного `ptr`, в EEPROM, начиная с `location`.

8.15.3. Доступ к EEPROM в реальном времени

Предшествующие макросы и функции ждут перед возвратом, пока EEPROM прочитается или запишется. Файлы `rteeprom.h` и `rteeprom.c` в каталоге `\icc\examples.avr` содержат исходный текст для процедур, которые читают и записывают в EEPROM, но не ждут перед возвратом, пока завершится аппаратная операция. Это особенно полезно для среды многозадачного режима реального времени. Функция “`ready`” предусмотрена, чтобы гарантировать, что операция завершена. Это особенно полезно для функции записи в EEPROM, так как запись в EEPROM, может отнять длительное время.

8.16. Обращение относительных вызовов и переходов

В устройствах с 8 КБ памяти программ, все области памяти достижимы командами относительного перехода и вызова (`rjmp` и `rcall`). Чтобы выполнить это, относительные переходы и вызовы обращаются вокруг границы 8 КБ. Например, прямой переход по байтовому адресу 0x2100 (0x2000 – 8 КБ) обращается в байтовый адрес 0x100.

Эта опция автоматически обнаруживается менеджером проекта всякий раз, когда объем целевой памяти программ равен точно 8192 байтам.

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

Первый параметр передается в регистрах R16/R17, если он типа `integer` и в регистрах R16/R17/R18/R19, если он типа `long` или число с плавающей точкой. Второй параметр передается в R18/R19, если это возможно. Все остальные параметры передаются через программный стек.

При отсутствии прототипа функции, целочисленные параметры меньшие, чем `int` (например, `char`) должны повышаться к типу `int`. Если прототип функции доступен, стандарт Си оставляет решение реализации компилятора. ICCV7 для AVR не повышает тип параметра, если прототип функции доступен. Если регистры используются для передачи параметра-байта, используются оба регистра, но старший регистр не определен. Например, если первый параметр – байт, оба регистра R16/R17 будут использоваться с регистром R17, содержащим неопределенное значение. Однобайтовые параметры, передаваемые через программный стек, также занимают 2 байта. Мы можем изменить это поведение и упаковывать байтовые параметры плотнее в какой-либо будущей версии.

Если R16/R17 используется, чтобы передать первый параметр, а второй параметр имеет тип `long` или `float`, младшая половина второго параметра передается через R18/R19, а старшая половина через программный стек.

Значения `integer` возвращаются в регистрах R16/R17, а `long`, и `float` возвращаются в R16/R17/R18/R19. Байтовые значения возвращаются в R16 с неопределенным R17.

9.2.3. Предохраняемые регистры

Ассемблерные функции должны сохранять и восстанавливать следующие регистры:

- R28/R29 или Y (это – указатель кадра)
- R10/R11/R12/R13/R14/R15/R20/R21/R22/R23

Эти регистры называются предохраняемые регистры, так как их содержимое не должно изменяться при вызове функции. Вы можете настроить компилятор, чтобы он не использовал регистры R20, R21, R22, R23 и тогда вы не должны сохранять и восстанавливать эти четыре регистра. См. 8.9. Глобальные регистры.

9.2.4. Volatile регистры

Другие регистры:

- R0/R1/R2/R3/R4/R5/R6/R7/R8/R9/R24/R25/R26/R27/R30/R31
- SREG

могут использоваться в функциях без сохранения и восстановления. Эти регистры называются изменяющиеся регистры (volatile), так как их содержание может быть изменено вызовом 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 изменяющиеся регистры, так как нормальные функции Си не предохраняют их. Так как обработчик прерывания функционирует асинхронно по отношению к нормальным операциям программы, обработчик прерывания и функции, которые он вызывает, не должны изменять никакие машинные регистры. Исключение – когда вы устанавливаете режим, чтобы компилятор не использовал регистры R20, R21, R22, и R23. Тогда обработчик прерывания может использовать эти четыре регистра непосредственно.

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 типа 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`, возвращаемые функцией, находятся в одном и том же наборе регистров R16-R19.

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 представляется 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. Указатели на функции

Для полной совместимости с функцией компрессора кода, все косвенные ссылки на функции должны производиться с помощью дополнительного уровня косвенности. В Си это выполняется автоматически, если вы вызываете функцию, используя указатель на функцию. Другими словами, указатели на функции ведут себя, как ожидается, за исключением того, что немного медленнее.

Следующий пример на ассемблере иллюстрирует это:

```
; assume _foo is the name of the function
.area func_lit
PL_foo:: .word _foo ; create a function table entry
.area text
ldi   R30,<PL_foo
ldi   R31,>PL_foo
rcall xicall
```

Вы можете использовать библиотечную функцию `xicall`, чтобы вызвать функцию косвенно после помещения адреса из таблицы входов в функции в пару R30/R31. Таблица входов в функции размещается в специальной области, называемой `func_lit`. См. 9.7. Области программы.

9.5. Машинные процедуры Си

Большинство операций Си транслируется в прямые инструкции AVR. 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-разрядных и 32-х разрядных данных
- `mpy, div, mod, neg, rmpy, rdiv, rmod, ...` – процедуры для 32-х разрядных длинных целых и чисел с плавающей запятой

9.6. Использование памяти программ и данных

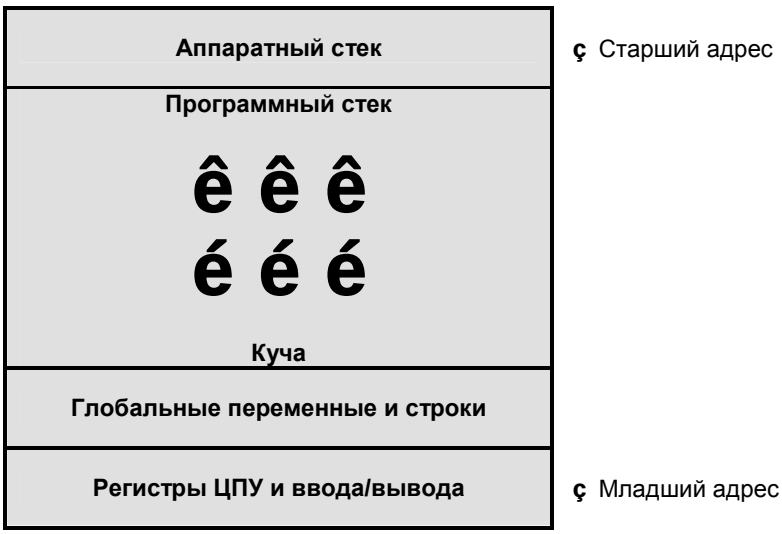
9.6.1. Память программ

Память программ используется для хранения кода вашей программы, таблиц констант и начальных значений для некоторых данных типа строк и глобальных переменных. См. 8.2. Данные программы и память констант. Компилятор генерирует образ памяти в форме выходного файла, который может использоваться пакетами программ, такими как ISP (внутрисхемный программатор).

В настоящее время, компилятор никак не использует память программ выше 64 КБ, кроме как для кода программы. Для обращения к памяти выше границы 64 КБ (как в устройствах Mega 103), вы должны вызвать команду ELPM непосредственно после установки регистра RAMPZ.

9.6.2. Внутренняя память данных SRAM

Память данных используется для сохранения переменных, кадров стека и кучи для динамического распределения памяти. В общем случае, они не появляются в выходном файле, но используются, когда программа выполняется. Программа использует память данных следующим образом:



Нижняя часть карты памяти имеет адрес 0. Первые 96 (0x60) адресов – регистры ЦПУ (CPU) и ввода/вывода (IO). Более новые устройства ATmega имеют даже большее количество регистров ввода/вывода. Компилятор помещает глобальные переменные и строки после регистров ввода/вывода. Выше переменных – область, где вы можете располагать динамическую память. См. 7.7. Стандартная библиотека и функции памяти. В верхних адресах, в конце SRAM располагается аппаратный стек. Ниже него – программный стек, растущий вниз. Программист должен гарантировать, что аппаратный стек не врастет в программный стек, и программный стек не врастет в кучу. Иначе все кончится непредсказуемым поведением программы. См. 8.6. Стеки.

9.6.3. Внешняя память данных SRAM

Если вы выбираете целевое устройство с 32 КБ или 64 КБ внешней памяти SRAM, то стеки размещаются наверху внутренней памяти SRAM и растут вниз к нижним адресам памяти. Память данных начинается над верхней частью аппаратного стека и растет вверх. Распределение выполнено по-другому, потому что внутренняя память SRAM имеет более быстрый доступ, чем внешняя память SRAM и в большинстве случаев более выгодно расположить стеки в более быстрой памяти.

9.6.4. Верхние 32 КБ внешней памяти данных SRAM

В редких случаях, когда вы имеете 32 КБ внешней памяти SRAM, и это – верхние 32 КБ адресного пространства, вы можете использовать это, выбирая “Internal SRAM Only” в разделе 4.13. Опции компилятора: Целевой контроллер, и затем добавить специальный параметр -bdata:0x8000.0xFFFF в панели редактирования “Other Options”. Для быстродействия стеки будут расположены во внутренней памяти SRAM, а глобальные данные будут расположены во внешней памяти SRAM.

9.7. Области программы

Компилятор размещает сгенерированный код и данные в различных областях. См. 12.4. Директивы ассемблера. Области, используемые компилятором, описаны здесь в порядке возрастания адреса памяти.

9.7.1. Память только для чтения

- `func_lit` – область таблиц функций. Каждое слово в этой области содержит адрес входа в функцию. См. 9.4. Указатели на функции и 12.1. Компрессор кода.
- `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.д.
- `text` – эта область содержит код программы.

9.7.2. 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.

9.7.3. Память EEPROM

- `eeprom` – эта область содержит данные EEPROM. Данные EEPROM записываются в `<output file>.eep` как Intel HEX файл независимо от формата выходного файла.

Работа компоновщика состоит в том, чтобы собрать однотипные области из всех входных объектных файлов и объединить их вместе в выходном файле. См. 12.6. Операции компоновщика.

9.7.4. Области, создаваемые программистом

Если вы используете `#pragma text / data / lit / abs_address` чтобы назначить ваши собственные области памяти, вы должны самостоятельно гарантировать, что их адреса не накладываются на адреса, используемые компоновщиком. Попытка наложения адресов может заставить, а может и не заставить компоновщик генерировать ошибку, поэтому вы должны всегда 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`. **Имена файлов с расширением `.lis` – выходные листинги ассемблера, не содержащие полной информации и не должны использоваться.** Файл листинга содержит ассемблерный код вашей программы, сгенерированный компилятором, с включениями исходного текста Си, машинным кодом и адресами. Значения не включенных данных и библиотечный код показывается только в зарегистрированной версии.

Однако, даже с этими ограничениями, файл листинга неоценим для отладки вашей программы, если вы не имеете средств отладки, которые понимают формат COFF. Некоторые дешевые Внутрисхемные Эмуляторы (ICE) также могут использовать файл листинга для управления отладкой в дополнение или вместо COFF.

10.2. Отладка COFF-кода и работа в AVR Studio

Если вы выбрали формат выходного файла COFF, отладочная информация исходного уровня доступна для COFF-совместимых отладчиков типа AVR Studio, который вы можете свободно загрузить с веб-сайта Atmel по адресу http://www.atmel.com. **Обратите внимание, что из-за ограничений формата COFF и AVR Studio 3.X, все исходные файлы и выходной COFF-файл должны быть в одном и том же каталоге, если вы используете AVR Studio 3.X.** Эти ограничения отсутствуют в AVR Studio 4.X.

10.2.1. Использование AVR Studio

Не используйте AVR Studio для вызова компилятора ICCV7 for AVR. Начиная с версий 3.5 и 4.X AVR Studio, эта возможность не работает. Вместо этого, пожалуйста, используйте следующую процедуру:

1. Создайте проект, используя среду ICCAVR.
2. Удостоверьтесь, что 4.12. Опции компилятора: Компилятор установлены для получения COFF или COFF/HEX. При использовании AVR Studio 3.X, убедитесь, что не определили "Выходной каталог" с ограничениями AVR Studio 3.X.
3. Скомпилируйте ваш проект, используя ICCV7 for AVR. Вы получите `<file>.cof` в каталоге проекта, где `<file>` – имя вашего проекта.
4. Откройте `<file>.cof` в AVR Studio. Это откроет исходный код файлов вашего проекта.

10.2.2. Использование терминала с AVR Studio

Для использования окна терминала в AVR Studio 3.X в режиме симулятора, вы должны сделать следующее:

1. Скопировать файл `\iccv7avr\libsrc.avr\iostudio.s` в каталог вашего проекта.
2. Установить 4.12. Опции компилятора: Компилятор, выбрав "AVR Studio Compatible IO".

Теперь симулятор будет использовать окно терминала для обмена по UART.

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 имеющимся 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. Обработка каждого файла зависимостей выполняется рекурсивно перед попыткой обработки 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 целевой файл.

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 для 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 решить 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 компиляции

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 AVR IDE на случай, если вы захотите управлять компилятором, используя ваш собственный редактор-среду типа Codewright. Все параметры передаются драйверу, а драйвер в свою очередь передает соответствующие параметры дальше.

Общий формат команды:

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

где `iccavr` – имя драйвера компилятора. Как видите, вы можете вызывать драйвер с множеством файлов, и драйвер выполнит операции со всеми файлами. По умолчанию, драйвер затем скомпонует все объектные файлы вместе, чтобы создать выходной файл.

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` – соответствующий компилятор. Например, `-Wf-Matmega`.
- `-Wa` – ассемблер.
- `-Wl` – (буква el) - компоновщик.

11.4.1. Параметры драйвера

- `-c` – только компиляция файла в объектный файл (не вызывается компоновщик).
- `-o <name>` – имя выходного файла. По умолчанию, имя выходного файла такое же, как имя входного файла, или такое же, как у первого входного файла, если вы передаете список 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, следующие опции должны использоваться с префиксом `-Wf-`, например `-Wf-str_in_flash`.

- `-Mavr_mega` – генерировать команды ATmega, такие как `call` и `jmp` вместо `rcall` и `rjmp`.
- `-Mavr_enhanced` – генерировать расширенные основные команды и `call` и `jmp`.
- `-Mavr_enhanced_small` – генерировать расширенные основные команды кроме `call` и `jmp`.
- `-str_in_flash` – располагать символьные строки только в памяти FLASH.
- `-use_elpm` – генерировать `ELPM` вместо `LPM` для доступа к FLASH памяти. Это полезно для программы загрузчика для устройств с большей, чем 64 КБ Flash памятью.
- `-r20_23` – не использовать регистры от R20 до R23 для генерации объектного кода. Полезно, если вы хотите зарезервировать эти регистры как глобальные (См. 8.9. Глобальные регистры).

11.4.4. Параметры ассемблера

- `-W` – включить обращение адресов. См. 8.16. Обращение относительных переходов и вызовов. При применении драйвера, вы должны использовать `-Wa-W`.
- `-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 старта 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 по умолчанию. Если файл задан только именем без информации о пути, то он должен быть размещен в библиотечном каталоге.
- `-W` – включить обращение адресов. См. 8.16. Обращение относительных переходов и вызовов. Заметьте, что вы должны использовать префикс `-Wl`, так как драйвер не знает об этой опции непосредственно (например, `-Wl-W`).
- `-fihx_coff` – выходными форматами являются и COFF и Intel HEX.
- `-fcoff` – выходным форматом является COFF. Обратите внимание, что, если вы имеете данные EEPROM, то они всегда записываются в формате Intel HEX в файл `<output>.eep`.
- `-fintelhex` – выходным форматом является `Intel HEX`.

- `-dram_end:<address>` – определяет конец внутренней области памяти RAM. Это используется для размещения областей памяти. См. 7.2. Файл запуска, чтобы инициализировать значение аппаратного стека. Для классических не Mega устройств, `ram_end` – это размер SRAM плюс 96 байт регистров ЦПУ и ввода/вывода минус 1. Для Mega устройств, это размер SRAM минус 1. Внешняя SRAM не влияет на это значение, так как аппаратный стек всегда располагается во внутренней RAM для улучшения быстродействия.
- `-dhwstk_size:<size>` – определяет размер аппаратного стека. Аппаратный стек располагается наверху SRAM, а программный стек под ним. См. 8.6. Стеки.
- `-l<lib name>` – компоновать со специфицированными библиотечными файлами в дополнение к `libcavr.a` по умолчанию. Это может использоваться, чтобы изменить поведение функции в `libcavr.a`, так как `libcavr.a` всегда компонуется последним. `Lib name` – имя библиотечного файла без префикса `lib` и без суффикса `.a`. Например:

```
-lstudio "libstudio.a"  использовать с AVR Studio IO
-llpavr "liblpavr.a"    использовать полную printf
-lfpavr "libfpavr.a"    использование printf с плавающей запятой
```

- `-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. Это полезно если вы пишете чисто ассемблерное приложение.
- `-S0` – генерировать формат COFF, совместимый с AVR Studio 3.X.
- `-S1` – генерировать формат COFF, совместимый с AVR Studio от 4.00 до 4.05.
- `-S2` – генерировать формат COFF, совместимый с AVR Studio 4.06+.

Определение 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. Так как попытка наложения памяти может заставить или не заставить компоновщик генерировать ошибку, вы всегда должны сами проверять файл карты `.mp` (используйте IDE 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. 9.7. 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.

- `-bfunc_lit:<address ranges>` – назначение адресных интервалов для области `func_lit`. Формат: `<start address>[<.end address>]`, где `address` – байтовые адреса. Любая память, не используемая этой областью, будет использована областями, которые следуют за ней, и это объявляет действительный размер FLASH памяти. Например, некоторые типичные значения:

```
-bfunc_lit:0x60.0x10000 для ATmega
-bfunc_lit:0x1a.0x800       для 23xx
-bfunc_lit:0x1a.0x2000      для 85xx
```

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

```
-bdata:0x60. 0x800          для ATMega
-bdata:0x60. 0x80           для 23xx
-bdata:0x60. 0x200          для 85xx
```

- `-beeprom:<address ranges>` – назначение адресных интервалов для EEPROM. Данные EEPROM записываются в `<output file>.eep` как Intel HEX файл независимо от формата выходного файла.

12. ИНСТРУМЕНТАРИЙ

12.1. Компрессор кода

Code Compressor (tm) – компрессор кода – оптимизатор современного уровня, 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 вмешивается в традиционную оптимизацию 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 кода не влияет на отладку на уровне исходного кода с AVR Studio.

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. 9.7. Области программы. Это выполняется автоматически, если вы используете Си.

12.1.4. Временная дезактивация компрессора кода

Иногда вы можете пожелать временно отключить компрессор кода. Например, возможно код чрезвычайно чувствителен ко времени, и не может себе позволить затраты из-за потери рабочих циклов, проходя через дополнительные вызовы и возвраты из функций. Вы можете делать это кодовыми фрагментами с парой команд заключенными в скобки:

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

Компрессор кода игнорирует команды во фрагменте между этими директивами ассемблера.

Включаемый файл `macros.h` содержит два новых определения для использования в Си программе:

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

12.2. Система управления версиями

ПРОФЕССИОНАЛЬНАЯ версия пакета среды разработки предоставляет набор инструментальных средств управления конфигурацией и интерфейс для управления вашим исходным кодом. Утилиты командной строки Revision Control System (RCS) – это утилиты системы управления версиями, использующие соглашения GNU (см. 1.13. Благодарности для замечаний по программному обеспечению GNU). RCS контролирует множественные изменения исходных файлов, позволяя вам при необходимости просматривать старейшие версии файлов. Среда предоставляет простой интерфейс к RCS, который является достаточным для наиболее общих задач. Чтобы выполнять 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 GNU RCS).

12.2.1. Репозиторий RCS

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

Каждое изменение файла имеет номер изменения и опциональную метку. Вы ссылаетесь на специфическую версию по номеру или метке. Метка полезна для сохранения кадра специфического набора изменений (например, перед тем, как вы выпускаете ваше программное обеспечение).

При расширенном использовании, вы можете даже изменять позднее изменение и “объединять” в ваших изменениях, или иметь множественные изменения для того же самого файла, сделанного различными людьми и согласовывать различные изменения (если не имеется конфликтов). Тема расширенного использования здесь не обсуждается.

12.2.2. Файлы Checkin и Checkout

Чтобы добавить новую версию файла в репозиторий, вы используете команду регистрации checkin (утилита `ci`). Чтобы изменить файл в репозитории, вы используете команду поверки checkout (утилита `co`). В самом простом случае, специальная опция к `ci` регистрирует файл и затем выполняет checkout непосредственно так, чтобы вы могли продолжать модифицировать файл.

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

12.3. Синтаксис ассемблера

12.3.1. Word и Byte операнды и оператор ` (backquote)

В AVR FLASH-память программ адресуется, как слова, если рассматривается как инструкции программы, или как байты, если используется как таблица только для чтения. Таким образом, в зависимости от используемых команд, операнды, содержащие адреса памяти программ, могут обрабатываться как адреса байта или слова.

Для однозначности ассемблер ICCV7 for AVR всегда использует адрес байта. Некоторые команды, например JMP и CALL, неявно преобразуют адрес байта в адрес слова. Чаще всего это выполняется прозрачно, если вы используете символические метки как операнды JMP/CALL. Однако если вы используете числовой адрес, то вы должны определить его как адрес байта. Например,

```
jmp 0x1F000
```

переходит к слову по адресу 0xF800. Когда вы должны определить адрес слова (например, при использовании директивы `.word`), можно использовать оператор ` (backquote):

```
PL_FUNC::
.word `func
```

помещает адрес слова `func` в слово по адресу глобальной метки `PL_func`.

Ассемблер имеет следующий синтаксис.

12.3.2. 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 – синонимы. Имя является или именем символа, который является постоянным значением, или именем метки, которая является значением программного счетчика (PC) на данный момент. Имя может иметь длину до 30 символов. Имена чувствительны к регистру, исключая мнемоники команд и директивы ассемблера.

12.3.3. Видимость имен

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

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.

12.3.4. Числа

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

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

12.3.5. 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 или пробелами или символами 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-м столбце строки.

12.3.6. 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.

12.3.7. Команды

Командой может быть команда AVR, директива ассемблера или макро-вызов. Поле операндов содержит параметры команды. Эта страница не описывает команды AVR, так как ассемблер использует стандартные мнемоники Atmel. Используйте документацию Atmel с описаниями команд. Исключения:

- `xcall` – применимо только к mega устройствам, которые поддерживают длинный вызов или команду перехода. Транслируется или в `rcall` или в `call`, в зависимости от расположения метки.
- `xjmp` – применимо только к mega устройствам, которые поддерживают длинный вызов или команду перехода. Транслируется или в `rjmp` или в `jmp`, в зависимости от расположения метки.

12.3.8. 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 значением программного счетчика. 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 и `foo` и `bar` – оба являются внешними символами.

12.3.9. 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 другие операторы должны применяться к константам или символам разрешимым ассемблером (как символам, определенным в файле).

Обратите внимание, чтобы получить старший и младший байт выражения, вы используете операторы > и <, а не операторы `high()` и `low()` ассемблера Atmel.

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

12.3.10. “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 компоновщику. 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 байт, область text_1
      .area data    <- 10 байт
      .area text    <- 20 байт, область text_2
  file2.o:
      .area data    <- 20 байт
      .area text    <- 40 байт, область text_3
  ```

  В этом примере, text_1, text_2, и так далее – имена областей, используемые в этом примере. Практически, они – не данные индивидуальные имена. Положим, что начальный адрес области text установлен в нуль. Затем, 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, пока начальный разделитель соответствует конечному разделителю. Между разделителями допустимы любые печатные символы ASCII и следующие символы в стиле Си, каждый из которых начинается с наклонной чертой 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 NULL (\0) в конец последовательности символов. Приемлемо 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 AVR. Обратите внимание, что `.long` может иметь только постоянные 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 (до встреченного первым) будет ассемблирован. Условные выражения могут иметь вложение до 10 уровней. Пример:

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

  Значение из a загрузится в R10, если символ cond не равен нулю, и значение из b загрузится в R10, если cond равно нулю.

- **.include** "<filename>"

  Обрабатывает содержимое файла, специфицированного 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 между 0 и 9, заменяется соответствующим параметром макрокоманды, когда макрокоманда вызывается. Вы не можете определять имя макрокоманды, которое находится в противоречии с мнемоникой команды или директивой ассемблера. См. .endmacro и макро вызовы. Например, следующее определяет макрокоманду с именем 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 в значение 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 команды Atmel AVR. Большинство из них использует тот же самый синтаксис, что и ассемблер Atmel. Пожалуйста, обратитесь к документации Atmel для получения полной информации относительно системы команд. Более новые AVR используют некоторые новые команды, которые не доступны в оригинальных AVR. Грубо говоря, существует три семейства команд AVR: классический AVR (например, 8515), оригинальный MegaAVR (например, Mega103), и новейший расширенный MegaAVR (например, Mega48). Некоторые команды, типа CALL и JMP, являются доступными только для устройств с большим, чем 8 КБ объемом памяти кода. Об операторах старших и младших байтов, см. 12.3.9. Операторы.

Арифметические и логические команды		
ADD Rd,Rr	ADC Rd,Rr	ADIW Rdl,K (a)
SUB Rd,Rr	SUBI Rd,K (b)	SBC Rd,Rr
SBCI Rd,K	SBIW Rdl,K	AND Rd,Rr
ANDI Rd,K	OR Rd,Rr	ORI Rd,K
EOR Rd,Rr	COM Rd	NEG Rd
SBR Rd,K	CBR Rd,K	INC Rd
DEC Rd	TST Rd	CLR Rd
SER Rd		
megaAVR		
MUL Rd,Rr	MULS Rd,Rr	MULSU Rd,Rr
FMUL Rd,Rr	FMULS Rd,Rr	FMULSU Rd,Rr
Команды переходов		
RJMP label	IJMP	JMP label
RCALL label	ICALL	CALL label
RET	RETI	CPSE Rd,Rr
CP Rd,Rr	CPI Rd,K	SBRC Rr,b
SBRS Rr,b	SBIC P,b	SBIS P,b
BRBS s,label	BRBC s,label	BRxx label (c)
Расширения megaAVR		
EIJMP	EICALL	
Команды передачи данных		
MOV R,dRr	LDI Rd,K	
LD Rd,X; X+; -X	LD Rd,Y; Y+; -Y	LDD Rd,Y+q
LD Rd,Z; Z+; -Z	LDD Rd,Z+q	LDS Rd,label (d)
ST X,Rr; X+; -X	ST Y,Rr; Y+; -Y	STD Y+q,Rr
ST Z,Rr; Z+; -Z	STD Z+q,Rr	LPM
ELPM	IN Rd,P	OUT P,Rr
PUSH Rr	POP Rd	

megaAVR		
MOVW Rd,Rr (e)	LPM Rd,Z; Z+	ELPM Rd,Z; Z+
SPM		
Команды работы с битами		
SBI P,b	CBI P,b	LSL Rd
LSR Rd	ROL Rd	ROR Rd
ASR Rd	SWAP Rd	BSET s
BCLR s	BST Rr,b	BLD Rd,b
<flag sets> (f)	<flag clears> (g)	
Команды управления микроконтроллером		
NOP	SLEEP	WDR
BREAK		
Псевдокоманды ассемблера ImageCraft		
XCALL label (h)	XJMP label (i)	

(a). ADIW/SBIW Rdl: R26, R28, R30
(b). xxI – инструкции прямого операнда: Rd должен быть R16-R31.
(c). BRxx где xx один из EQ, NE, CS, CC, SH, LO, MI, PL, GE, LT, HS, HC, TS, TC, VS, IE, ID. Это синонимы "BRBS s" и "BRBC s"
(d). Используйте >label для старшего байта и <label для младшего байта.
(e). Rd и Rr должны быть четными
(f). SEC, SEZ, SEI, SES, SEV, SET, SEH являются синонимами "BSET s"
(g). CLC, CLZ, CLI, CLS, CLV, CLT, CLH являются синонимами "BCLR s"
(h). Транслируется в RCALL, если ссылается внутри файла и в CALL в противном случае.
(i). Транслируется в RJMP, если ссылается внутри файла и в JMP в противном случае.

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. Создание файла запуска, который будет первым связываемым файлом. Файл запуска инициализирует среду исполнения, чтобы выполнить программу 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, которые вызваны непосредственно или косвенно, будут участвовать в связывании. Все определенные пользователем объектные файлы (например, ваши программные файлы) будут связаны.
3. Присоединение стандартной библиотеки Си `libcavr.a` к концу списка файлов.
4. Просмотр объектных файлов для поиска неразрешенных ссылок. Компоновщик отмечает объектный файл (возможно в библиотеке) который удовлетворяет ссылкам и добавляет к списку неразрешенных ссылок. Этот процесс повторяется до тех пор, пока не останется никаких ожидающих обработки неразрешенных ссылок.
5. Объединение всех отмеченных объектных файлов в выходной файл и генерация файлов карты и листинга.

Наконец, если это – УСОВЕРШЕНСТВОВАННАЯ или ПРОФЕССИОНАЛЬНАЯ версия и если опция оптимизации Code Compressor включена, то вызывается компрессор кода.

12.6.1. Распределение памяти

Поскольку компоновщик объединяет области кодов входных объектных файлов, он назначает им адреса памяти на основании диапазонов адресов, переданных из командной строки. (См. 11.4.5. Параметры компоновщика). Эти параметры в свою очередь обычно передаются из среды разработки на основании определенного устройства. То есть в обычном случае вы не должны делать что-нибудь дополнительно, и среда разработки с компилятором сами корректно распределяют память.

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. Попытка наложения адресов может заставить, а может и не заставить компоновщик генерировать ошибку. Вы должны всегда проверять файл карты `.mp` (используйте меню *View>Map File*) для поиска потенциальных проблем.

12.7. Отладочный формат ImageCraft

Файл отладки ImageCraft (расширение .dbg) – частный ASCII формат, который описывает отладочную информацию. Компоновщик генерирует “стандартный” выходной отладочный формат непосредственно в дополнение к этому файлу. Например, AVR компилятор генерирует файл формата 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

12.8. Библиотекарь

Библиотека – это коллекция объектных файлов в специальной форме, которую понимает компоновщик. Когда объектный файл компонента библиотеки вызван вашей программой непосредственно или косвенно, компоновщик извлекает из него библиотечный код и связывает его с вашей программой. Определяемая стандартом библиотека – `libcavr.a`, которая содержит стандартные функции Си и функции, специфичные для AVR. Другие библиотеки, такие как `libstudio.a`, переопределяют некоторые функции из `libcavr.a` для достижения различного поведения без изменения кода вашей программы. Например, определив связи с `libstudio.a`, ваша программа может использовать окно терминала в AVR Studio. Для вас процесс связывания обычно прозрачен – выбирая соответствующие 4.10. Опции компилятора, среда генерирует правильные ключи для программ компилятора.

Бывают случаи, когда вам необходимо изменить существующую или создать новую библиотеку. Для этого предусмотрен инструмент командной строки `ilibw.exe`.

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

12.8.1. Компиляция файла в библиотечный модуль

Каждый библиотечный модуль – это просто объектный файл. Следовательно, чтобы создать библиотечный модуль, вы должны скомпилировать исходный файл в объектный. Это может быть выполнено, открывая файл и вызывая 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 libcavr.a
```

12.8.3. Добавление или замена модуля

Добавление или замена модуля:

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, следующее заменяет функцию `putchar` в `libcavr.a` вашей версией:

```
cd \icc\libsrc.avr
<модифицировать putchar() в iochar.c>
<компилировать iochar.c в iochar.o>
copy \icc\lib\libcavr.a ; копировать библиотеку
ilibw -a libcavr.a iochar.o
copy libcavr.a \icc\lib ; копировать обратно
```

Команда `ilibw` создает библиотечный файл, если он не существует. Чтобы создать новую библиотеку, дайте `ilibw` новое имя файла библиотеки.

12.8.4. Удаление модуля

Ключ команды `-d` удаляет модуль из библиотеки. Например, следующее удаляет `iochar.o` из библиотеки `libcavr.a`:

```
cd \icc\libsrc.avr
copy \icc\lib\libcavr.a ; копировать библиотеку
ilibw -d libcavr.a iochar.o ; удалить
copy libcavr.a \icc\lib ; копировать обратно
```