Компилятор ANSI C и среда разработки для ARM

Версия 7.XX

IMAGECRAFT C COMPILER FOR ARM 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 7 ноября 2005 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 10 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 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 ARM 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 \iccv7arm\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:\iccv7arm\drivers`, заменив `c:\iccv7arm` 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. Имеются инсталляции, позволяющие лицензировать работу от одного до 50 пользователей.

Все ваши машины должны принадлежать одной сети. Вы должны назначить одну машину, как сервер 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 ARM на сервер ключа и на все машины, где вы желаете использовать данный продукт.

На сервере ключа:

2. Следуйте инструкциям, описанным выше в 1.3. Использование аппаратного ключа для инсталляции драйвера аппаратного ключа.
3. Запустите из командной строки программу `c:\iccv7arm\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:

4. Переименуйте `c:\iccv7arm\bin\darm.dll` в `c:\iccv7arm\bin\darm.dll.orig`.
5. Переименуйте `c:\iccv7arm\bin\darmnet.dll` в `c:\iccv7arm\bin\darm.dll`.
6. В директории Windows (например, `c:\windows` или `c:\winnt`), создайте файл по имени `iccARM.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:\iccv7arm\dongle_license`, вы должны написать:

   ```
   [dinkey]
   DinkeyNetPath=\\foo\c\iccv7arm\dongle_license
   ```

7. Когда вы запустите ICCV7 for ARM на клиентских машинах, 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 ARM`.
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 ANSI стандарт языка Си. Ее основные особенности:

- Интегрированная среда разработки (IDE) с интегрированным редактором и менеджером проекта предназначена для работы в 32-х разрядных версиях Windows. Исходные файлы организуются в проекты. Редактирование и создание 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 ARM” 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 создаем нашу репутацию, основанную на наилучшей поддержке. 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 послал 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 выслали нам ваши файлы проекта, так чтобы мы могли дублировать 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 весь проект по запросу, обычно достаточно, если вы сможете создать компилируемую функцию и выслать ее нам. Пожалуйста, не присылайте нам никакие файлы, если они не были запрошены.

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

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

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

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

Список рассылки не должен использоваться для общих вопросов поддержки. С другой стороны, наши активные заказчики 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. `libcarm.a` – базовая библиотека, содержащая стандартную библиотеку Си и специфичные для ARM процедуры. Компоновщик связывает программу с библиотечными модулями или файлами только при наличии ссылок на них. Вы можете создавать или изменять библиотеки при необходимости.

1.9.2. Выходные файлы

- `.s` – выходной ассемблерный файл. Он генерируется компилятором для каждого исходного файла Си.
- `.o` – объектный файл, получаемый трансляцией ассемблерного файла. Выходной исполнимый файл есть результат компоновки группы объектных файлов.
- `.hex` – выходной Intel HEX файл.
- `.s19` – исполнимый файл в формате Motorola/Freescale S19 Record.
- `.lst` – файл листинга с объектным кодом и конечными адресами вашей программы, собранными в один файл.
- `.mp` – файл карты, содержащий символьную информацию и размер вашей программы.
- `.dbg` – внутренний командный файл отладки ImageCraft.
- `.elf` – исполнимый файл ELF/DWARF с отладочной информацией, соответствующий спецификации DWARF/2.

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

1.10. Прагмы и расширения

1.10.1. #pragma

Компилятор распознает следующие директивы прагма:

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

  Объявляет функции как обработчики прерываний, чтобы компилятор генерировал инструкции возврата из прерывания вместо обычного возврата из функции и сохранял и восстанавливал все регистры, используемые функциями. Эта прагма должна предшествовать определениям функций. См. 8.6. Сброс и обработка прерываний.

- `#pragma language=extended`

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

- `#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 введения:

  ```
  #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. 8.4. 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.
- Объявление обработчика прерывания. Наши компиляторы используют прагму, чтобы объявить функцию как обработчик прерывания. 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 использование расширений, как нам кажется, является желание привязать заказчика к среде поставщика компилятора, чем обеспечить лучшее решение.

  Используя наш компилятор, верхний пример может быть переписан так:

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

- 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 ассемблерный код 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 в правильных адресах 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. следующий 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
  ```

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 процессором (микроконтроллером).

- 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 сколько 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, отслеживает содержание регистров и устраняет копии и другие подобные излишества.

1.12.1. 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 значения.

  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 мощный распределитель регистров усилен “сетью” (отличающейся от 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 вообще 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 значительное количество усилий 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.” Утилита Make – Jacob Navia. Попробуйте недорогой Win32 компилятор Jacob Navia на http://www.cs.virginia.edu/~lcc-win32.

Pride Embedded, LLC (http://www.PridEEmbedded.com/) помогли с кодом инициализации App Builder ARM и написали библиотеку PEC LPC210x. Посетите сайт ImageCraft, чтобы узнать больше о библиотеке PEC LPC210x и о коммерчески доступных готовых продуктах на базе контроллеров ARM созданных Pride Embedded. Sten Larsson из Nohau предоставил оригинальные заголовочные файлы ARM.

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 `\iccv7arm\bin`. 7 Zip использует GNU LGPL лицензию, и вы можете получить копию программы с сайта http://www.7-zip.org.

Весь код используется по разрешению. **Пожалуйста, сообщайте обо всех ошибках непосредственно нам.**

2. ВВЕДЕНИЕ

2.1. Быстрый старт

ICCV7 для ARM может использоваться для написания программ для любого процессора и микроконтроллера ARM7 (а с некоторой доработкой, даже для ARM9), так как все изготовители кристаллов используют то же самое обобщенное ядро ARM7. Однако различные микроконтроллеры ARM7 имеют различные карты памяти и могут требовать различных файлов запуска. См. 7.2. Файл запуска.

ICCV7 для ARM довольно легок в использовании. Однако чтобы упростить начало работы, в каталоге `c:\iccv7arm\examples.arm` есть ряд примеров проектов для популярной платы ARM7 MCU, такой как Philips LPC2106 на основе MicroBolt, Atmel SAM7S64-EK и т.д. Используйте команду среды разработки *Project>Open*, чтобы открыть один из типовых проектов, вызовите команду меню *Project>Build* (или нажмите на кнопку панели инструментов **"Build"**) и скомпилируйте проект.

2.1.1. Приложения MicroBolt

Кроме инсталлированных примеров, автор MicroBolt написал обширные заметки о приложениях и большое количество примеров программ, используя ICCV7 для ARM. Они могут быть найдены на сайте http://www.micromint.com/products/bolt.htm. Хотя на рынке имеется несколько плат на основе Philips LPC2106, и ICCV7 для ARM должен работать с любыми из них без проблем, из-за их обширной документации мы очень рекомендуем MicroBolt как платформу для разработки.

2.1.2. Старт нового проекта

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

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

2.2. Анатомия Си программы

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

2.2.1. Примеры

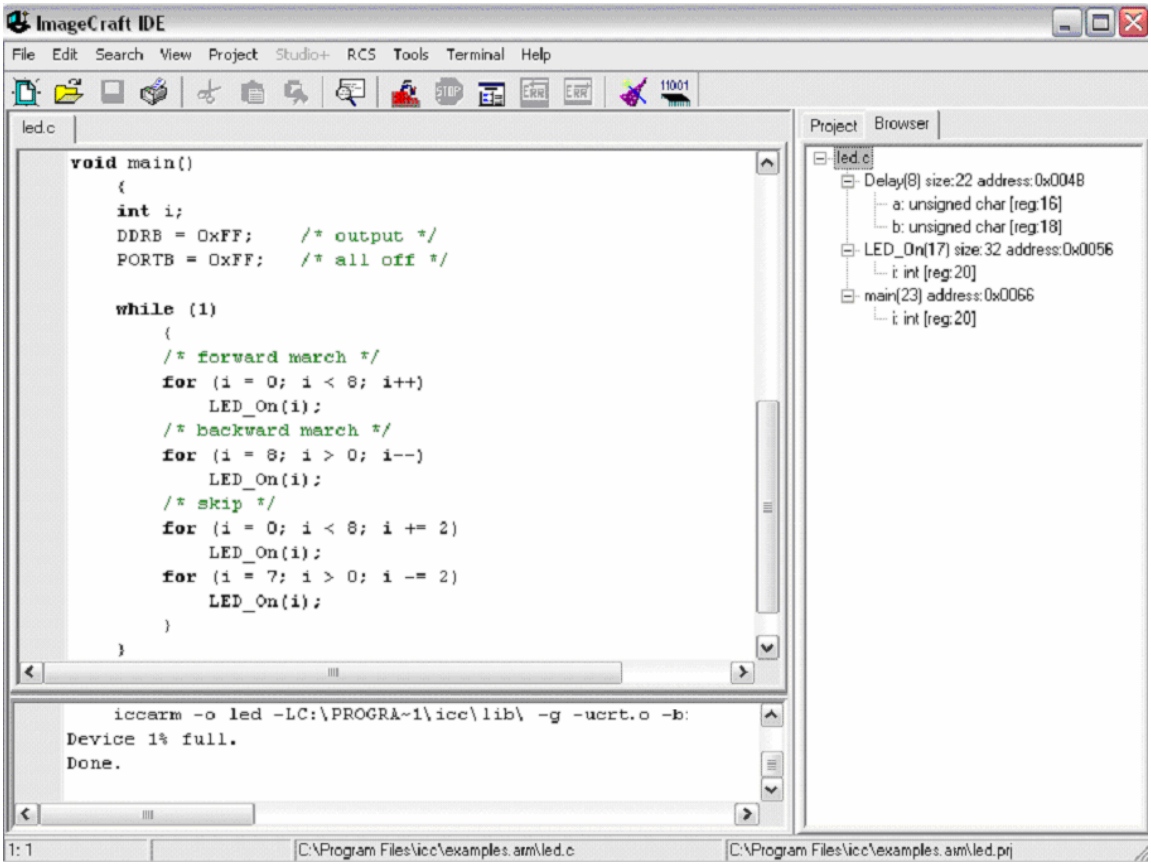
ICCV7 for ARM поставляется с набором библиотечных функций для кристалла Philips LPC210X. Они находятся в директории `c:\iccv7arm\examples.arm\pec.lib\` (замените `c:\iccv7arm` вашим путем установки). Некоторые свойства примеров включают:

- Полный демо проект для контроллера LPC2106 ARM.
- Множество Си и заголовочных файлов для упрощения понимания.
- Установки и инициализация PLL.
- Выбор функций и направления выводов ввода/вывода GPIO (показаны входы, выходы и установка выводов специальных функций).
- Установки и приоритизация Philips ARM VIC (векторный контроллер прерываний).
- Timer-0: Установки, инициализация, ISR (процедура обработки прерывания).
- Timer-1: Установки, инициализация, ISR.
- PWM-0: Установки, инициализация, ISR.
- UART-0: Установки, инициализация, выбор скорости, ISR по приему символа, сохранение и разбор строки, процедура передачи.
- External Interrupt-1: Установки, ISR, код конвертирования прерывания чувствительного к уровню в прерывание чувствительное к фронту (1 прерывание вместо нескольких).
- Процедуры задержки (измеренные на точность).

Свойства поддаются изменению, и может быть добавлено большее количество функций. Библиотека свободна для личного использования, и вы можете получить коммерческую лицензию за минимальную стоимость. Пожалуйста, посетите наш веб-сайт для получения подробностей.

2.3. Краткий обзор среды разработки

Главное окно среды разработки IDE разделено на три части:



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

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

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

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

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

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

2.4. Использование менеджера проекта

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

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

3. СРЕДА РАЗРАБОТКИ

3.1. Концепция среды разработки

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

3.2. Управление проектом

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

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

3.2.1. Создание нового проекта

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

Опции компилятора устанавливаются в меню *Project>Options*.

3.2.2. Опции проекта

Опции компилятора настраиваются в окне, вызываемом командой *Project>Options...*. и сохраняются вместе с файлами проекта, поэтому можно иметь различные проекты с различными целевыми контроллерами. Когда вы начинаете новый проект, используется заданный по умолчанию набор опций. Вы можете установить текущие опции как опции по умолчанию или загрузить опции по умолчанию в текущий набор опций. Опции по умолчанию сохраняются в файле `deficcarm.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 *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 правым щелчком 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. При выполнении компиляции менеджер проекта игнорирует файлы, отличные от файлов с исходным кодом.

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 маленькая и может храниться как одиночный файл. 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 добавьте 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 о состоянии среды разработки, такую, как состояние компиляции. Сообщения об ошибках компиляции начинаются с `!E...` и отмечаются маленьким красным значком на желобе слева. Щелчок на строке ошибки перемещает курсор на строку, вызвавшую ошибку, в окне редактора.

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

3.8. Эмулятор терминала

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

4. СИСТЕМА МЕНЮ

4.1. Всплывающие меню

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

4.2. Меню File

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

- **New** – создать вкладку с новым пустым окном редактора для ввода текста.
- **Reopen**... – открытие файла из списка недавно открывавшихся файлов.
- **Open...** – открыть существующий файл для редактирования.
- **Reload... from Disk** – отказаться от изменений и перезагрузить открытый файл с диска.
- **Reload... from Backup** – перезагрузить открытый файл из резервной копии. См. Save.
- **Save** – сохранить открытый файл на диске. Перед сохранением нового содержимого, опционально создается файл резервной копии `<file>.<~ext>`. См. 4.14. Опции среды разработки.
- **Save As...** – сохранить открытый файл на диске под новым именем.
- **Close** – закрыть активный редактируемый файл. Выводится запрос на запись несохраненных изменений.
- **Compile File... To Object** – компилировать открытый файл в объектный файл с расширением `.o`. Заметим, что объектный файл не является загружаемым в программатор или симулятор. См. 1.9. Типы и расширения файлов. Это полезно для проверки файла на ошибки, создания объектного файла для библиотеки, или чтобы создать новый файл запуска. См. 7.2. Файл запуска.
- **Compile File... To Output** – компилировать открытый файл в выходной, подходящий для загрузки в программатор или симулятор. Обычно, для управления файлами вашего проекта, используется 3.3. Список файлов проекта и окно обозревателя кода, но если проект небольшой, вы можете использовать эту команду, чтобы создать выходной файл. Используются заданные по умолчанию 4.10. Опции компилятора.
- **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 12.1. Система управления версиями.

4.7.1. RCS функции среды разработки

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

4.8. Меню Tools

- **Environment Options** – открыть диалоговое окно 4.14. Опции среды разработки и 4.15. Опции терминала.
- **Editor and Print Options** – открыть диалоговое окно 4.16. Опции редактора и печати.
- **Application Builder** – вызов Application Builder, который является утилитой, создающей процедуры инициализации ARM, 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 Выполнить 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 `\iccv7arm\bin\deficcarm.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 `\iccv7arm\include` (замените `\iccv7arm` на ваш путь установки) к путям включаемых файлов и вы не должны добавлять его явно.

- **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 `\iccv7arm\lib` (замените `\iccv7arm` 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 `libcarm.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.
- **Optimizations** – управляет уровнем и типом оптимизации из следующих вариантов:
  - **Enable Global Optimizations** – только в ПРОФЕССИОНАЛЬНОЙ версии. Вызывает глобальный оптимизатор размера кода и быстродействия программы.
- **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**".
- **Program Memory** – определяет стартовый адрес и объем памяти программ.
- **Data Memory** – определяет стартовый адрес и объем памяти данных SRAM.
- **PRINTF Version** – эта группа переключателей позволяет вам выбирать, с какой версией `printf` будет связана ваша программа. Больше возможностей потребует большее количество кода. См. 7.6. Стандартные функции ввода/вывода для подробностей:
  - Малая или Базовая версия: доступны только `%c`, `%d`, `%x`, `%X`, `%u`, и `%s` форматы без модификаторов.
  - Длинная: допускается длинный модификатор. В дополнение к полям ширины и точности, поддерживаются `форматы %ld, %lu, %lx`, и `%lX`.
  - С плавающей запятой: добавляются форматы `%e`, `%f` и `%g` для плавающей запятой.
- **Additional Libraries** – возможно использование библиотек, отличных от стандартных, поддерживаемых продуктом. 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, `rtos` относится к библиотечному файлу `librtos.a`. Все библиотечные файлы должны заканчиваться расширением `.a`.
- **Default Startup** – среда ассоциирует файл запуска (см. 7.2. Файл запуска) размещенный в библиотечном каталоге с каждым известным устройством. Имя файла отображается здесь. Файл запуска инициализирует программную среду прежде, чем управление передается программе пользователя. Прежде всего, из-за различий в реализации контроллера прерываний устройств ARM, различные семейства ARM почти всегда требуют собственные специальные файлы запуска.
- **Use Custom Startup File** – вы можете использовать ваш собственный файл запуска как один из файлов проекта, установив этот переключатель. Файл запуска должен удовлетворять требованиям, детализированным в разделе 7.2. Файл запуска.
- **No C$$init in Linker Command File** – по соглашению, векторы прерывания размещаются в области C$$init. См. 9.3. Области программ. Обычно, C$$init расположена в начале памяти FLASH (обычно по адресу 0). Если эта опция разрешена, среда не записывает расположение C$$init в 12.3.1. Командный файл компоновщика. Это позволяет вам или не использовать C$$init область или расположить ее по нестандартному адресу, используя “Дополнительный командный файл”. Обратите внимание, что, если вы не используете C$$init область, компоновщик выдаст предупреждение.
- **Unused ROM Fill Bytes** – заполняет неиспользуемые области ROM определенным целочисленным шаблоном.
- **Additional Command File** – среда генерирует 12.3.1. Командный файл компоновщика на основе выбранного устройства, чтобы описать карту памяти устройства для компоновщика. Вы можете определить дополнительный командный файл, например, если вы хотите разместить некоторые функции по специфическим адресам памяти. См. 9.3. Области программ. Обратите внимание, что адреса, которые вы используете в вашем командном файле, не должны конфликтовать с адресами в командном файле, сгенерированном средой разработки.

- **Stack Sizes (bytes)** – среда генерирует 12.3.1. Командный файл компоновщика, который помещает вершину стека в конце памяти данных SRAM. Это – стек FIQ. Указатели стека для стека IRQ и стеков супервизора располагаются на некоторое количество байт ниже этого значения. Если вы желаете ввести устройство в пользовательский режим (см. 8.5. Режимы и стеки процессора ARM), вы будете должны изменить 7.2. Файл запуска для определения пространства для стека пользователя ниже стека супервизора и изменить режим устройства перед вызовом функции `main`.
- **FIQ** – определяет число байт для стека FIQ. Вершина стека IRQ затем устанавливается на “вершину стека FIQ” (например, в верхней части SRAM) – “FIQ байты”.
- **IRQ** – определяет число байт для стека IRQ. Стек супервизора затем устанавливается на “вершину стека IRQ” – “IRQ байты”.

4.14. Опции среды разработки

Это диалоговое окно управляет общими установками среды разработки:

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

4.14.1. Опции просмотра обозревателя кода

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

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

4.14.2. Предпочтительный редактор

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

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

В окнах редактирования параметров команд, вы можете использовать `%f`, чтобы сослаться на имя файла и `%l` для ссылки на номер строки. Информация внешнего редактора сохраняется в файле `\iccv7arm\bin\editors.ini`. Мы обеспечиваем информацию для некоторых из наиболее популярных редакторов в файле `\iccv7arm\bin\editors.installed`. Когда вы устанавливаете программу в первый раз, среда копирует `editors.installed` в `editors.ini`. Когда вы обновляете программу впоследствии, файл `editors.ini` остается нетронутым, так что ваши изменения сохраняются. После обновления, вы можете открыть файл `editors.installed` и копировать и вставлять информацию любого нового редактора в файл `editors.ini`.

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

4.15. Опции терминала

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

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

4.16. Опции редактора и печати

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

4.16.1. Опции

Печать

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

Общее

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

Разное

- **Right Margin** – отображать правый отступ (серая вертикальная линия) в определенном столбце.
- **Gutter** – отображать желоб определенного размера. Обратите внимание, что желоб используется для отображения закладок, номеров строк и других элементов.

- **Block indent step size** – число шагов при использовании команд выравнивания Block Indent и Outdent.
- **TAB Columns** – определяет позиции столбцов табуляции. Если не определено, то для вычисления позиций табуляции используется значение “TAB stop”.
- **TAB stop** – число символов, используемое символом TAB, если не используется “TAB Columns".

4.16.2. Контекстная подсветка

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

4.16.3. Назначения клавиш

Страница позволяет вам изменять назначения клавиш для команд редактирования.

4.16.4. Шаблоны кода

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

5. ПРЕПРОЦЕССОР Си

5.1. Диалекты препроцессора Си

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

5.2. Предопределенные макросы

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

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

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

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

5.3. Поддерживаемые директивы

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

5.3.1. Макроопределения

- `#define macname definition`

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

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

  Функция-подобный макрос, позволяющий передавать параметры в макроопределение.

- `#undef macname`

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

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

5.3.2. Условная обработка

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

- `defined(name)`

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

- `#if <expr>`

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

- `#ifdef name / #ifndef name`

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

- `#elif <expr>`

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

- `#else`

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

- `#endif`

  Заканчивает условную компиляцию группы строк.

5.3.3. Дополнительно

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

  Обрабатывается содержимое указанного файла.

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

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

- `#error "message"`

  Выводит сообщение об ошибке.

- `#warning "message"`

  Выводит предупреждающее сообщение. Является расширением ImageCraft.

- `#pragma ...`

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

5.4. Строковые литералы и склейка лексем

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

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

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

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

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

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

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

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

```
foo ## bar
```

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

6. КРАТКОЕ ОПИСАНИЕ Си

6.1. Введение

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

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

6.1.1. Стандарты Си

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

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

C99 – последний стандарт Си. Пока некоторые стремились к созданию нового Си как подмножества C++, здравомыслие возобладало 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 объявления

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. 8.3. 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 вывод на устройство LCD. Исходный код библиотеки доступен, и его можно использовать как образец. Заменить заданную по умолчанию библиотечную функцию можно одним из следующих методов:

- Вы можете включить вашу функцию в один из ваших файлов проекта. В этом случае система компилятора не будет использовать библиотечную функцию. Обратите внимание, что в этом случае, в отличие от библиотечного модуля, ваша функция всегда будет включаться в выходной файл программы, даже если вы ее не используете.
- Вы можете создать вашу собственную библиотеку. Подробнее см. 12.4. Библиотеки.

7.2. Файл запуска

Ваш проект должен иметь файл запуска. Мы поддерживаем различные файлы запуска для различных семейств процессоров, т.к. они могут иметь различные требования. Файл запуска определяет глобальный символ `_cstart`, являющийся стартовой точкой вашей программы. Функции файла запуска:

- Установка векторов прерываний. См. 8.5. Режимы и стеки процессора ARM. Вход по сбросу определяется переходом на метку `_cstart`, являющимся точкой входа в Си.
- Резервирование памяти для некоторых глобальных переменных, используемых для целей установки системы Си. Это включает (но не ограничивается):

  ```
  $icc$ROM_ADDR
  $icc$RAM_ADDR
  $icc$DATA_SIZE
  $icc$URAM_ADDR
  $icc$UDATA_SIZE
  ```

  Это специальные переменные, известные компоновщику, и устанавливаемые им при компоновке всех объектных файлов проекта.

Код по адресу `_cstart` выполняет:

- Обнуление памяти данных в секции “Cudata”. Эта секция содержит глобальные неинициализированные данные Си программы, которые должны быть обнулены согласно требованиям стандарта Си. Стартовый адрес секции Cudata определяется переменной `$icc$URAM_ADDR`, а размер секции определяется переменной `$icc$UDATA_SIZE`.
- Копирование данных инициализированной памяти из `$icc$ROM_ADDR` в область `$icc$RAM_ADDR` размером в `$icc$DATA_SIZE` байт. Инициализированные глобальные переменные хранятся в памяти ROM FLASH и копируются в память RAM.
- Установка указателей стеков. См. 8.5. Режимы и стеки процессора ARM.
- Переход на метку `_main` – пользовательскую точку входа в Си программу.

В некоторых системах, вам может потребоваться выполнить дополнительную установку. Например, серия Atmel AT91x40 обычно требует выполнения "перепланировки карты памяти". Вам может также потребоваться выполнить это, если ваш выполнимый код находится в SRAM. См. документацию на устройство для выяснения подробностей.

7.2.1. Области файла запуска

Векторы прерывания должны находиться в области C$$init. Фактический код запуска должен быть в нормальной Си области кода C$$code. См. примеры файлов запуска для деталей.

7.2.2. Модификация файла запуска

Мы предоставляем несколько примеров файлов запуска: `crtat91.s` для Atmel SAM7S (без перепланировки памяти), `crtat91eb40a.s` для оценочной платы Atmel EB40A и `crtlpc2k.s` для серии Philips LPC2K. Если вы используете другие устройства ARM7 с интегрированной памятью FLASH и SRAM, для которых мы не имеем пока файла запуска, часто вы будете должны сделать изменения только в векторах IRQ файла запуска. Если имеется что-нибудь специфическое для процессора, вероятно, вы сможете сделать это в коде пользователя, возможно даже на Си. Для расширенных устройств ARM (например, ARM7 с внешней памятью или ARM9), файл запуска может быть более сложен, поскольку может потребоваться установка контроллера памяти и других возможностей процессора. Файл `crtat91eb40a.s` поддерживает опциональную перепланировку памяти и может быть хорошей отправной точкой для устройств, требующих такой поддержки.

Файлы `crtlpc2k.s` и `crtat91.s` созданы объединением файла таблицы векторов `crt???vec.s` (где ??? это lpc2k или at91 соответственно) и обобщенного Си кода запуска `crtbody.s`. Эти файлы находятся в каталоге исходных кодов `c:\iccv7arm\examples.arm`.

Если вы создаете файл запуска для еще не поддерживаемого процессора ARM и желаете совместно использовать его с другими людьми, пожалуйста, свяжитесь нами.

7.3. Общее описание библиотеки Си

Исходный текст библиотеки (`c:\iccv7arm\libsrc.arm\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 \iccv7arm\libsrc
unzip -s libsrc.zip
```

7.3.1. Другие заголовочные файлы

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

`assert.h` – макрос диагностики.

`ctype.h` – функции символьного типа.

`float.h` – характеристики формата плавающей точки.

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

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

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

`stddef.h` – стандартные определения.

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

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

`string.h` – функции манипулирования строками.

7.4. Функции символьного типа

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

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

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

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

  Возвращает не нуль, если c – алфавитный символ.

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

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

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

  Возвращает не нуль, если c – цифра.

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

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

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

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

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

  Возвращает не нуль, если c – печатный символ.

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

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

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

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

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

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

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

  Возвращает не нуль, если c – шестнадцатеричная цифра.

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

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

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

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

7.5. Математические функции с плавающей точкой

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

- `float` **`asinf`**`(float x)`

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

- `float` **`acosf`**`(float x)`

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

- `float` **`atanf`**`(float x)`

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

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

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

- `float` **`ceilf`**`(float x)`

  Возвращает наименьшее целое число не меньшее чем `x`.

- `float` **`cosf`**`(float x)`

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

- `float` **`coshf`**`(float x)`

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

- `float` **`expf`**`(float x)`

  Возвращает `e` в степени `x`.

- `float` **`exp10f`**`(float x)`

  Возвращает `10` в степени `x`.

- `float` **`fabsf`**`(float x)`

  Возвращает абсолютное значение `x`.

- `float` **`floorf`**`(float x)`

  Возвращает наибольшее целое число не большее чем `x`.

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

  Возвращает остаток от `x/y`.

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

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

- `float` **`froundf`**`(float x)`

  Округляет `x` до самого близкого целого числа.

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

  Возвращает `x*2**exp`.

- `float` **`logf`**`(float x)`

  Возвращает натуральный логарифм `x`.

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

  Возвращает логарифм `x` по основанию `10`.

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

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

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

  Возвращает `x` в степени `y`.

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

  Возвращает квадратный корень `x`.

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

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

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

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

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

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

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

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

Данные функции работают с аргументом и возвращаемым значением типа `float`. Соответствующие функции с аргументом и возвращаемым значением типа `double` имеют те же имена, но без суффикса `f`.

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:\iccv7arm\examples.arm\` для устройств UART. Вы можете начать с одного из примеров и добавить его в ваш список файлов проекта.

7.6.1. Вывод возврата каретки

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

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

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

7.6.2. Использование Printf с несколькими устройствами

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

7.6.3. Список стандартных функций ввода/вывода

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

  Возвращает символ из UART, используя режим опроса.

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

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

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

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

  Спецификаторы формата являются подмножеством стандартных форматов:

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

Флаги формата:

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

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

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

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

`0` – заполнять нулями вместо пробелов.

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

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

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

d – печатать следующий параметр как десятичное целое число

o – печатать следующий параметр как восьмеричное целое число без знака

x – печатать следующий параметр как шестнадцатеричное целое число без знака

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

u – печатать следующий параметр как десятичное целое число без знака

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

c – печатать следующий параметр как символ ASCII

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

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

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

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

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

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

  Печатать строку, сопровождаемую NL.

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

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

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

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

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

  - `[l]` – длинный модификатор. Этот опциональный модификатор определяет, что соответствующий параметр имеет тип указатель на длинное целое
  - d – ввод – десятичное число. Параметр должен быть указателем на (`long`) `int`.
  - `x/X` – ввод – шестнадцатеричное число, возможно начинающееся с `0x` или `0X`. Параметр должен быть указателем на (`long`) `int` без знака.
  - u – ввод – десятичное число. Параметр должен быть указатель на (`long`) `int` без знака.
  - – 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 _SetHeapSize перед использованием любой из процедур распределения памяти (calloc, malloc, и realloc).

- `int` **`abs`**`(int i)`

  Возвращает абсолютное значение i.

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

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

- `double` **`atof`**`(const char *s)`

  Преобразовывает строку s в double и возвращает его.

- `long` **`atol`**`(char *s)`

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

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

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

- `void` **`exit`**`(status)`

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

- `void` **`free`**`(void *ptr)`

  Освобождает предварительно выделенную память кучи.

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

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

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

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

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

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

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

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

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

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

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

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

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

- `void` **`_SetHeapSize`**`(unsigned heap_size)`

  Инициализирует кучу для процедур выделения памяти. `malloc` и связанные процедуры управляют памятью в области кучи. См. 9.4. Карта памяти относительно выделения памяти. `_SetHeapSize` устанавливает начало кучи в конец области Cudata (т.е. в конец глобальных пользовательских переменных и т.п.) и конец кучи в `start + heap_size` (в байтах).

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

  Возвращает псевдослучайное число между 0 и `RAND_MAX`.

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

  Перераспределяет предварительно выделенный фрагмент памяти с новым размером.

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

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

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

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

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

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

7.8. Строковые функции

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

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

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

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

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

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

  Копирует `n` байт из `s2` в `s1`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `size_t` **`strlen`**`(char *s)`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.9. Функции с переменными параметрами

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

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

  Инициализирует переменную foo.

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

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

- **va_end**(va_list foo)

  Заканчивает обработку переменных параметров.

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

```
#include <stdarg.h>

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

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

8. ПРОГРАММИРОВАНИЕ ARM

8.1. Доступ к специфическим ресурсам ARM

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

В дополнение к заголовочным файлам стандартной библиотеки Си, продукт поставляется с заголовочными файлами, специфичными для устройств ARM. Они определяют регистры ввода/вывода так, чтобы вы могли использовать их в ваших программах на Си. Например:

```
#include <arm_macros.h>
#include <philips/lpc210x.h>
...
#pragma interrupt_handler TimerMatch
void TimerMatch(void)
    {
    changeLed();
    T0_IR = 0xFF; // clr Timer interrupt
    VICVectAddr = 0xFF;
    }
```

В этом примере, заголовочный файл `lpc210x.h` в подкаталоге `philips` определяет регистры ввода/вывода `T0_IR` и `VICVectAddr`, так что вы можете использовать их в вашей программе точно так же как идентификаторы Си.

Вы можете использовать Application Builder, чтобы генерировать код инициализации, включая обработку прерываний для периферийных устройств ARM. См. 3.6. Application Builder.

8.2. Специфические заголовочные файлы ARM

Специфические для ARM заголовочные файлы размещены в каталогах `c:\iccv7arm\include\<vendor>\` (замените `c:\iccv7arm` путем инсталляции) где `<vendor>` является одним из производителей устройств ARM, в настоящее время являющийся одним из: `atmel`, `motorola`, `oki`, `philips`, `sharp`, и `st`. Будут добавляться и другие поставщики и устройства.

Заголовочный файл `arm_macros.h` определяет некоторые полезные для программирования ARM макросы, такие как:

```
__DISABLE_INTERRUPT()
__ENABLE_INTERRUPT()
```

8.2.1. Схемы доступа к регистрам ввода/вывода

Различные поставщики компиляторов определяют регистры ввода/вывода, используя различные методы. Мы используем стандартные битовые маски Си и операторы косвенного доступа к памяти, так что наши заголовочные файлы переносимы на другие компиляторы Си. Однако заголовочные файлы других производителей могут использовать их собственные расширения и таким образом могут быть непереносимы в нашу среду (и в среду некоторых других производителей).

8.3. 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 поразрядные операторы, которые являются особенно полезными.

- **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.

8.4. Встроенный ассемблер

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

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

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

```
unsigned int val;

asm("mov R0,%val\n"
    "SWI");
```

Этим способом можно сослаться на любой символ Си, исключая метки оператора Си `goto`. Вообще, использование встроенного ассемблерного кода для ссылок на локальные регистры ограничено: возможно, что никакие регистры не будут являться доступными, если вы объявили слишком много регистровых переменных в функции. В таком случае, вы получили бы ошибку ассемблера.

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

8.5. Режимы и стеки процессора ARM

Процессор ARM может функционировать в семи различных режимах: User (Пользователь), FIQ (Быстрое прерывание), IRQ (Нормальное прерывание), Supervisor (Супервизор), Abort (Аварийное завершение), Undefined (Неопределенное) и System (Система). Все режимы за исключением User являются *привилегированными*, означая, что в них имеется полный доступ к ресурсам системы. Все режимы за исключением User и System являются режимами *исключения*, означая, что они обычно вызываются исключительной ситуацией, произошедшей в системе. Следующая таблица описывает режимы процессора ARM.

Режим	Привилеги-рованность	Указатель стека	Вызывается
FIQ	да	R13_fiq	исключением (FIQ)
IRQ	да	R13_irq	исключением (IRQ)
Supervisor	да	R13_svc	исключением (сброс и SWI)
Abort	да	R13_abt	исключением (предвыборка или авария)
Undefined	да	R13_und	исключением (неопределенная инструкция)
System	да	R13	изменением битов режима CPSR
User	нет	R13	изменением битов режима CPSR

Если процессор работает в режиме Пользователя, он не может поменять режим на другой кроме как через исключение. При сбросе, процессор ARM входит в режим Супервизора. Для малых ядер ARM типа ARM7TDMI, обычно находящихся в микроконтроллерах, лучше всего оставить процессор работать в режиме Супервизора, чтобы он мог иметь доступ ко всем ресурсам системы. В других случаях, например, если вы хотите работать в защищенной OS, процессор должен быть настроен так, чтобы OS выполнялась в режиме Системы, а пользовательские задачи выполнялись в режиме Пользователя. Чтобы включить выполнение системы не в режиме Супервизора, вы должны изменить и перекомпилировать 7.2. Файл запуска.

Все режимы используют собственный указатель стека, кроме режимов Пользователь и Система, которые совместно используют один и тот же указатель стека. Следовательно, даже в настройке по умолчанию в режиме Супервизора, должны быть инициализированы два указателя стека: один для режима Супервизора и один для режима IRQ, если система должна обрабатывать прерывания. Заданный по умолчанию файл запуска устанавливает FIQ на верхний адрес SRAM, а стеки IRQ и Супервизора располагаются на некоторое количество байтов ниже. Размеры стеков FIQ и IRQ управляются в 4.13. Опции компилятора: Целевое устройство. Пространство Стека используется компилятором, прежде всего для сохранения и восстановления регистров. Поэтому, если ваш обработчик прерывания не вызывает больше чем приблизительно от двух до четырех вложенных подпрограмм, то объем около 100 байт был бы достаточен для стека IRQ. Конечно, если быть консервативным, вы можете захотеть выделить больший стек для IRQ, т.к. если случится переполнение стека, ваша программа, скорее всего, потерпит сбой.

8.6. Сброс и обработка прерываний

Архитектура ARM определяет мало источников прерывания. В частности имеется только один вектор для "нормальных" прерываний. Следующая таблица описывает прерывания:

Тип	Адрес обработчика по умолчанию
Сброс	0x0
Неопределенная инструкция	0x4
Программное прерывание (SWI)	0x8
Авария предвыборки данных	0xC
Авария данных	0x10
(не используется)	0x14
IRQ	0x18
FIQ	0x1C

По умолчанию, векторы расположены в адресном пространстве от 0x0 до 0x1F. Так как каждый вектор представляет одиночную инструкцию ARM, обычно производится просто переход к “быстрому” обработчику, размещенному по адресам от 0x20 до 0x3F, который делает некоторую обработку и, в случае необходимости, переходит к некоторым реальным обработчикам в нормальной области программы. Таким образом, в соответствии с соглашением и по определению архитектуры, область от 0x0 до 0x3F зарезервирована для обработки прерываний.

Схема с одиночным входом IRQ имеет особое место в мире микроконтроллеров, потому что микроконтроллер обычно имеет много периферийных устройств, что требует высочайшей эффективности обработки прерываний. В типичной реализации микроконтроллеров ARM (например, семейства Philips LPC и Atmel SAMS7), устройство имеет систему векторного контроллера прерываний, где пользователь назначает различные приоритеты периферийным прерываниям. Одиночный ARM IRQ вызывает обработчик, который делает быстрый векторный переход к обработчику пользователя для частной обработки. Эта схема обеспечивает гибкость ценой более трудного использования, по крайней мере, по сравнению с простым 8- или 16-разрядным микроконтроллером. Также различные производители имеют различные конструкции своих контроллеров прерываний.

Таким образом, типичная цепочка прерываний выглядит так:

- В файле запуска (См. 7.2. Файл запуска) вектор IRQ установлен для перехода к обработчику вектора системы, определенному устройством ARM. Например, вход выглядит так:

```
location 0x18:
Philips LPC     LDR PC,[PC,#-0xFF0]
Atmel AT91SAM7  LDR PC,[PC,#-0xF20]
```

- Установить векторный контроллер прерывания, чтобы указать на ваш обработчик прерывания. Контроллер крайне специфичен для устройства. Это может включать установку приоритета, установку адреса обработчика прерывания в таблице векторов и разрешение битов прерывания в контроллере.
- Разрешить регистр прерываний для периферийного устройства в пространстве регистров ввода/вывода.
- Разрешить прерывания на уровне процессора ARM, используя `__ENABLE_INTERRUPT()` если имеется `#include "arm_macros.h"`. Это всего лишь вызов встроенного ассемблерного кода для очистки правильных бит в регистре состояния системы CPSR.

- Когда происходит прерывание, процессор исполняет обработчик прерываний, который переходит к обработчику векторного контроллера прерываний и затем к вашему обработчику. Все они выполняются, используя команды перехода, а не BL переход и инструкцию связи, так что регистр связи R14 сохраняется. Правильный адрес возврата на четыре байта меньше чем значение R14.
- Если вы позволяете вложенные прерывания (например, позволяя более высокоприоритетному прерыванию прерывать ваш обработчик), то вы должны сохранить SPSR и модифицировать CPSR, чтобы снова разрешить прерывания.
- Перед возвратом, ваш обработчик должен сбросить регистр периферийных прерываний и состояние прерывания векторного контроллера прерываний. Для сброса состояния контроллера прерываний, сначала запретите прерывание в CPSR, если вы разрешили его ранее. Это восстановит сохраненный SPSR и осуществит возврат:

```
__DISABLE_INTERRUPT();
// Modify the vectored interrupt controller informing it
// that the interrupt has been handled.
...
// restore SPSR and return
```

Пожалуйста, обратитесь к документации на устройства конкретных производителей за точной информацией об их схеме обработки прерываний. Эти шаги относительно усложнены, особенно по сравнению с простой архитектурой прерываний 8- и 16-разрядных микроконтроллеров типа Atmel AVR. К счастью, вы можете писать ваши обработчики прерываний на Си, и компилятор выполнит некоторые из этих шагов за вас.

Процессор ARM также обеспечивает быстрое прерывание (FIRQ) для источника прерывания высокого приоритета. FIRQ имеет собственный вектор в таблице векторов ARM. Как и нормальное прерывание IRQ, контроллеры прерываний различных производителей обрабатывают FIRQ по-разному. Пожалуйста, обратитесь к документации на устройства производителей за деталями.

8.6.1. Си обработчики прерываний

Некоторые процессоры ARM поддерживают автоматическое сохранение и восстановление регистров их контроллерами прерываний. С такими контроллерами, назначьте адрес обработчика (который может быть только адресом функции Си) вектору и запрограммируйте контроллеры для автоматического сохранения и восстановления некоторых регистров.

Для большинства существующих контроллеров ARM7, тем не менее, обработчики прерываний нуждаются в специальном коде входа и выхода, отличающемся от нормальных функций Си. Тем не менее, вы можете писать обработчики прерывания на Си. В файле, где вы определяете функцию, перед определением функции вы должны сообщить компилятору, что функция является обработчиком прерывания, используя прагму:

```
 #pragma interrupt_handler <name> *
```

Например:

```
#pragma interrupt_handler timer0 timer1
...
void timer0(void)
    {
    ...
    }

void timer1(void)
    {
    ...
    }
```

Вы можете поместить несколько имен в одну прагму `interrupt_handler`, разделяя их пробелами. Используя прагму `interrupt_handler` для пролога функции, компилятор генерирует код, вычитающий 4 байта из R14 и сохраняя все используемые функцией регистры, включая “volatile” регистры. См. 9.2. Интерфейс ассемблера и соглашения о вызовах.

Можно использовать функции `__ENABLE_INTERRUPT()` и `__DISABLE_INTERRUPT()` для модификации состояния прерываний системы как того требует программа. В коде выхода из функции генерируется следующее:

```
ldmfd R13!,{<reglist>,PC}^
```

Обратите внимание на оператор ^ в конце списка регистров. Это устанавливает S бит в инструкции так, чтобы, когда PC был загружен, CPSR будет восстановлен из SPSR. Если вы должны изменить или сохранить SPSR, например, разрешая вложенные прерывания, вы можете использовать встроенный ассемблер:

```
#pragma interrupt_handler timer0
void timer0(void)
    {
    unsigned myspsr;

    asm("mrs %myspsr,spsr");// save SPSR
    __ENABLE_INTERRUPT();
    // nested interrupt may happen here
    ...
    __DISABLE_INTERRUPT();
    // modify vectored controller
    ...
    // restore myspsr
    asm("msr spsr,%myspsr");
    }
```

В будущем средства сохранения должны использовать 3.6. Application Builder для установки деталей прерывания, если он поддерживает ваше устройство ARM.

8.7. Перестройка карты памяти

Некоторые устройства ARM позволяют вам перестраивать карту памяти SRAM на адрес 0. Это позволило бы выполняющейся программе динамически изменять таблицу векторов, расположенную по адресам от `0x0` до `0x20`. Обычно вы делаете это, выдавая команду `REMAP` контроллеру памяти устройства в начальной части файла запуска Си. В более новых моделях ARM, контроллеры прерывания устройств устраняют большинство причин использовать эти средства.

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	4	0..4294967295
(signed) int	4	-2147483648..2147483647
pointer	4	0..4294967295
unsigned long	4	0..4294967295
(signed) long	4	-2147483648..2147483647
float	4	-1.175e-38..3.40e+38
double	8	-2.2250738e-308..1.7976931e+308

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

Тип `float` использует 32-х битный формат стандарта IEEE с 8-битной экспонентой, 23-х битной мантиссой, и 1-битным знаковым разрядом. Тип `double` использует 64-х битный формат стандарта IEEE с 11-битной экспонентой, 52-х битной мантиссой, и 1-битным знаковым разрядом.

9.2. Интерфейс ассемблера и соглашения о вызовах

Компилятор транслирует исходные файлы Си в ассемблерные файлы, которые затем обрабатываются ассемблером. Этот раздел объясняет соглашения, используемые компилятором, чтобы вы могли писать ассемблерные модули, совместимые с кодом, сгенерированным Си. Вообще, наш компилятор использует соглашение APCS (ARM Procedure Calling Standard – стандарт вызова процедур ARM).

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 до 1024 значащих символов. Чтобы сделать ассемблерный объект глобальным, вы должны использовать директиву `EXPORT`.

Например, объявление в ассемблерном файле:

```
EXPORT _var32

AREA "Cudata", DATA
_var32: SPACE 4
```

в файле Си соответствует следующему объявлению:

```
extern unsigned int var32;
...
```

9.2.2. Регистры аргументов и возвращаемых значений

Параметры Си передаются слева направо. С некоторыми исключениями, ICCV7 for ARM использует соглашение APCS и передает первые 16 байтов параметров в регистрах от R0 до R3. Вызываемая функция может использовать эти регистры для своих целей без сохранения и восстановления. Они известны как *volatile* (*изменчивые*) регистры. Возвращаемое значение находится в R0, если оно равно 4 байтам или меньше, или в R0/R1 для двойного слова. Любые параметры свыше первых 16 байтов передаются через стек. Структура всегда передается через стек.

Когда компилятор генерирует код для функции, возвращающей структуру, он создает временную область в стеке вызываемой функции и передает функции адрес этой области. Вызываемая функция записывает возвращаемый результат в эту временную область. Все эти действия прозрачны для пользователей.

9.2.3. Volatile регистры

Дополнительно к регистрам R0 – R3, функция может использовать R12 без сохранения и восстановления его значения.

9.2.4. Сохраняемые регистры

Если функция использует регистры R4 – R11, то она обязана сохранять их значения на входе в функцию и восстанавливать на выходе. Они известны как *preserved* (*сохраняемые*) регистры.

9.2.5. Соглашения об использовании регистров

Регистры R0 – R3 используются для передачи параметров функции или для вычисления выражений. Если компилятор должен использовать большее количество регистров для вычисления выражений в функции, он начнет использовать регистр R4 и следующие. Любые регистры от R4 до R10, не используемые для вычисления выражений назначаются для локальных переменных, используя усовершенствованный алгоритм распределения регистров. Несколько локальных переменных могут быть помещены в одни и те же регистры, если компилятор определяет, что их времена жизни не накладываются.

Регистр R11 используется как указатель кадра. Локальные переменные, не помещенные в регистры, параметры, передаваемые на стек, и другие временные области стека используются по ссылке при помощи указателя кадра. Регистр R12 – рабочий регистр, используемый для вычисления некоторых выражений.

Регистр R13 служит указателем стека и не должен модифицироваться пользователем непосредственно.

R14 является регистром связи для вызова функций.

R15 является программным счетчиком – PC.

9.3. Области программ

Компилятор генерирует код и данные в различных областях:

- `C$$code`

  Объявление области: `AREA "C$$code",CODE,READONLY`

  Эта область содержит программный код, литералы, строки и т.д.

- `Cidata`

  Объявление области: `AREA "Cidata",DATA`

  Эта область данных содержит инициализированные переменные. Фактические значения инициализации сохраняются в кодовой секции только для чтения, и Файл Запуска копирует значения из раздела кода в области RAM (См. 7.2. Файл запуска).

- `Cudata`

  Объявление области: `AREA "Cudata",DATA`

  Эта область данных содержит неинициализированные переменные. Файл запуска обнуляет эту область согласно требованиям Стандарта Си.

Вы можете создавать ваши собственные области. Например, вы можете захотеть разместить функцию в специфической области памяти (например, для более быстрого доступа). Вы можете сделать это, помещая функцию в области с другим именем и затем определить адрес этой области в командном файле компоновщика:

```
#pragma text:myarea
void func(void) { ... }
#pragma text:text
// must use "text:text" to reset the name
```

В командном файле компоновщика (`foo.cmd`):

```
MEMORY { myrom(RO): o = 0x300000 l = 0x200 }
SECTIONS { myarea > myrom }
```

Смотрите 12.3.1. Командный файл компоновщика для деталей формата файла. Вы определяете командный файл в 4.13. Опции компилятора: Целевое устройство. Обратите внимание, что адреса, которые вы используете в вашем командном файле, не должны конфликтовать с адресами в командном файле, генерируемом средой разработки.

Наконец, в соответствии с соглашением, файл запуска помещает векторы прерывания в область `C$$init`. По умолчанию длина этой области равна `0x40` байт и состоит из `0x20` байтов для векторов архитектуры ARM и из `0x20` байтов для обработчиков прерывания, которые умещаются в небольшом объеме памяти.

9.4. Карта памяти

Точная карта памяти зависит от конкретного устройства ARM. Например, в то время как большинство устройств ARM имеет FLASH память в нижней области адресного пространства, некоторые устройства используют функцию REMAP, чтобы назначить быструю память RAM по нижним адресам так, чтобы вы могли легко изменять векторы прерывания.

Большинство микроконтроллеров имеет больше памяти FLASH чем SRAM. Обычно их соотношение составляет от 5 до 20. Расположение регистров ввода/вывода зависит от устройства. С 32-разрядным адресным пространством (4 гигабайта), адресное пространство обычно раздроблено. Некоторые устройства ARM размещают векторы прерывания в верхней памяти. Проконсультируйтесь с техническим описанием на устройство для подробностей.

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”.
- Компилятор может делать кое-что непредвиденное даже притом, что это правильно. Например, для RISC-подобных процессоров типа Atmel AVR, TI MSP430 и ARM CPU, компиляторы могут помещать разные локальные переменные в тот же самый машинный регистр, пока использование локальных переменных не накладывается. Это значительно улучшает сгенерированный код, даже притом, что это может удивлять при отладке. Например, если вы помещаете окно часов в две переменные, которые компилятором размещаются в одном и том же регистре, обе переменные изменялись бы даже притом, что ваша программа изменяет только одну из них.
- Машинно-Независимый Оптимизатор делает отладку даже более спорной. MIO может устранять или перемещать код или изменять выражения, а для RISC-подобных процессоров, распределитель регистров может назначать различные регистры или адреса памяти одной и той же переменной в зависимости от места использования. К сожалению, в настоящее время большинство отладчиков имеют только ограниченную поддержку отладки оптимизированного кода.
- Вы можете столкнуться с ошибкой компилятора. Если вы сталкиваетесь с сообщением об ошибке в форме

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

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

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

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

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

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

10.1.1. Тестирование логики программы

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

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

10.1.2. Файл листинга

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

10.2. Отладка в исходном коде Си

ICCV7 for ARM генерирует файлы двух типов для отладки на уровне исходного кода Си:

1. ImageCraft DBG. Этот тип файла имеет расширение `.dbg`. Это основанный на ASCII и очень простой формат, разработанный для отладки на уровне Си. В настоящее время отладчик Nohau EMULARM Seehau напрямую поддерживает этот формат.
2. ELF/DWARF. Этот тип файла имеет расширение `.elf`. Формат соответствует документу DWARF 2.0, опубликованному UNIX International и одобренному TIS (Tool Interface Standard) для поколения dwarf. Этот формат принят почти всеми отладчиками и эмуляторами ARM.

Так как ELF/DWARF является стандартом, мы настоятельно рекомендуем производителям отладчиков и эмуляторов связаться с нами относительно прямой поддержки нашего формата DBG, так как это – очень простой формат, и легко обеспечить максимальную совместимость между продуктами.

11. КОМПИЛЯТОР КОМАНДНОЙ СТРОКИ

11.1. Процесс компиляции

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

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

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

Все эти шаги поддерживаются драйвером компилятора. Вы даете 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. Утилита make жалуется, если вы используете пробелы вместо символа табуляции. Каждый зависимый файл может быть целью в make-файле. Сопровождение каждого зависимого файла выполняется рекурсивно перед попыткой поддержки текущего целевого файла. Если, после обработки всех зависимостей, целевой файл оказался отсутствующим или более старым, чем любой из файлов зависимостей, make выполняет прилагаемые команды или неявным образом перекомпилирует его.

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

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

11.2.1. Параметры утилиты Make

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

Вы можете также определить макрос make в командной строке определением:

```
macro=value
```

11.3. Драйвер

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

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

11.4. Параметры компилятора

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

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`.
- `-Wa` – ассемблер.
- `-Wl` – (буква el) - компоновщик.

11.4.1. Параметры драйвера

- `-c` – только компиляция файла в объектный файл (не вызывается компоновщик).
- `-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.

11.4.4. Параметры ассемблера

- `-V` – показать информацию о версии и выйти.
- `-l` – генерировать ассемблерный файл листинга.
- `-o <file>` - задать имя выходного файла.
- `-?|-help` – вывести справочную информацию.

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).
- `-g` – генерировать информацию для отладки. Файл отладки имеет расширение .DBG. ICCV7 для ARM также генерирует `.elf` файл с информацией ELF/DWARF.
- `-u<crt>` – использовать `<crt>` вместо файла старта. Если файл задан только именем без информации о пути, то он должен быть размещен в библиотечном каталоге.
- `-V` – показать информацию о версии и выйти.
- `-o<name>` – задать имя выходного файла.
- `-l<libname>` – компоновать со специфицированными библиотечными файлами. `libname` – имя библиотечного файла без префикса `lib` и без суффикса `.a`.
- `-cf:<file>` – читать определение сегмента памяти из командного файла. Могут быть определены несколько командных файлов, но дублирующие определения из последних файлов игнорируются.

12. ИНСТРУМЕНТАРИЙ

12.1. Система управления версиями

ПРОФЕССИОНАЛЬНАЯ версия программного обеспечения предоставляет набор инструментов управления конфигурацией и интерфейс среды разработки для управления вашим исходным кодом. Программное обеспечение командной строки GNU Revision Control System (RCS) – это утилиты системы управления версиями (см. 1.13. Благодарности для замечаний по программному обеспечению GNU). RCS контролирует множественные версии исходных файлов, позволяя вам при необходимости просматривать 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 по GNU RCS).

12.1.1. Репозиторий системы управления версиями

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 разработки 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 снимка 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, сделанные различными людьми и согласовывать различные изменения (если не имеется конфликтов). Темы расширенного использования в этом документе не обсуждается.

12.1.2. Добавление и изменение файлов репозитория

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.

12.2. Ассемблер

Наш ассемблер использует синтаксис подобный ассемблеру инструментов компилятора ARM Inc. Текущая версия главным образом поддерживает компилятор Си, и команды Thumb не были тестированы.

12.2.1. Формат исходного кода

Каждая исходная строка должна быть в форме:

```
{symbol} {instruction|directive} {; comment }
```

Любое поле опционально. Однако команды не могут начинаться в первом столбце и должны предваряться символом или пробелом. Обычно `symbol` является меткой.

12.2.2. Предопределенные имена

Следующие имена не чувствительны к регистру символов и являются предопределенными: R0-R15, PC (R15), LR (R14), SP (R13), CPSR и SPSR.

12.2.3. Директивы

Ассемблер распознает следующие директивы:

12.2.4. ALIGN <expr>

`<expr>` должно быть степенью двойки. Перемещает счетчик программ для выравнивания по выражению `<expr>`. Промежутки при необходимости заполняются нулями.

12.2.5. AREA name {,attr}

`attr` может быть одним из `CODE/DATA` или `READONLY/READWRITE`. `CODE` подразумевает `READONLY`, а `DATA` подразумевает `READWRITE`.

12.2.6. CODE16

Переход к инструкциям Thumb.

12.2.7. CODE32

Переход к инструкциям ARM (по умолчанию).

12.2.8. DCB <expr> {,<expr>}

Определяет последовательность байтов. `<expr>` является либо числовой константой, либо строкой в кавычках. В отличие от Си, сгенерированный код строки не завершается нулем.

12.2.9. DCD <expr> {,<expr>}

Выравнивает счетчик программ по границе 32-битного слова и определяет последовательность 32-битных слов. `<expr>` – либо числовая константа, либо метка.

12.2.10. DCI <expr> {,<expr>}

`DCI` аналогичен `DCD` за исключением того, что работает в пространстве кода CODE.

12.2.11. DCW <expr> {,<expr>}

Выравнивает счетчик программ по границе 16-битного полуслова и определяет последовательность 16-битных полуслов. `<expr>` – числовая константа.

12.2.12. END

Сообщает ассемблеру, что был достигнут конец файла.

12.2.13. EXPORT label {,label}

Объявляет метки (определенные в данном модуле) которые являются видимыми в других модулях.

12.2.14. IMPORT label {,label}

Объявляет метки, которые определены в других модулях так, что на них можно ссылаться из данного модуля.

12.2.15. SPACE <expr>

Резервирует `<expr>` байт памяти без инициализации их значений.

12.2.16. Инструкции

Ассемблер распознает стандартные инструкции ARM. Полное руководство, конечно же, находится в ARM Inc.: http://www.arm.com/documentation/Instruction_Set/.

12.2.17. Предопределенные области ассемблера

Некоторые области имеют специальное значение для ассемблера и компоновщика. Они используются компилятором Си:

- `CODE,READONLY` – программный код и другие значения только для чтения, типа строк и таблиц помещаются в область READONLY CODE. В соответствии с соглашением, компилятор Си и компоновщик используют имя `C$$code` для заданной по умолчанию области программы.

  ```
  AREA C$$code,READONLY
  ```

- `DATA,RERADWRITE RAM` – область для инициализированных данных:

  ```
  AREA Cidata,READWRITE
  ```

  Вы можете использовать директивы ассемблера, такие как DCB и DCD, чтобы инициализировать символы, определенные в `Cidata` области. Так как значения RAM не сохраняются между перерывами питания, когда программа перезапускается, компоновщик сохраняет инициализированные значения в `C$$code` памяти. Одна из задач 7.2. Файл запуска состоит в копировании инициализированных значений в RAM. В соответствии с соглашением, компилятор Си и компоновщик используют имя `Cidata` для области инициализированных данных. Это место, где размещаются глобальные и статические инициализированные переменные Си.

- `DATA,NOINIT RAM` – область для неинициализированных данных:

  ```
  AREA Cudata,READWRITE
  ```

  Глобальные переменные без инициализированных значений помещаются в область `Cudata`, используя директиву `SPACE`. В соответствии с требованием Стандартного Си, они должны быть инициализированы нулями. Это выполняется в 7.2. Файл запуска. В соответствии с соглашением, компилятор Си и компоновщик используют имя `Cudata` для области неинициализированных данных. Это место, где размещаются неинициализированные глобальные и статические переменные Си.

12.3. Компоновщик

Основное назначение 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 деталями, шаги компоновки включают:

1. Добавление 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 в связывании. Все определенные пользователем объектные файлы (например, ваши программные файлы) будут связаны.
2. Просмотр объектных файлов для поиска неразрешенных ссылок. Компоновщик отмечает объектный файл (возможно в библиотеке) который удовлетворяет ссылке и добавляет к списку неразрешенных ссылок. Этот процесс повторяется до тех пор, пока не останется ожидающих обработки неразрешенных ссылок.
3. Объединение всех отмеченных объектных файлов в выходной файл и генерация файлов карты и листинга если необходимо.

Формат объектного файла произведенного ассемблером является вариантом формата COFF. Компоновщик производит файл в формате Intel HEX, файл карты `.MP`, а также файл отладки `.DBG`, если в командной строке определен ключ `-g`. При необходимости драйвер `iccarm` вызывает программу `DBG2DWARF`, чтобы создать исполняемый файл ELF/DWARF.

12.3.1. Командный файл компоновщика

Конечное размещение различных областей `AREA` определяется командным файлом компоновщика. Среда разработки генерирует файл, автоматически основанный на выборе, который вы делаете в 4.13. Опции компилятора: Целевое устройство. Обычно командный файл компоновщика состоит из трех частей: директивы `MEMORY`, директивы `SECTIONS`, и группы определений символов.

Директива MEMORY

Директива `MEMORY` определяет разные области памяти, доступные в целевом устройстве:

```
MEMORY
{
  <name 1>(<attr>): o = <starting address>, l = <length>
  ...
  <name n>: o = <starting address>, l = <length>
}
```

`<name>` дает имя области памяти и больше не имеет никакого специального значения. Каждая строка определяет область памяти, ее начало и длину. Числа могут быть десятичные или шестнадцатеричные, если они начинаются с `0x` или `0X`. Все участки памяти должны иметь уникальные имена и не должны накладываться. `(<attr>)` может быть либо `(RO)` либо `(RW)`, означая память, доступную Только для Чтения или для Чтения и Записи соответственно. Например, типичный командный файл может выглядеть следующим образом:

```
MEMORY
{
  ROM(RO): o = 0x200100 l = 0x002f00
  RAM(RW): o = 0x203000 l = 0x001000
  STARTUP(RO): o = 0x200000 l = 0x000100
}
SECTIONS
```

```
{
  C$$init > STARTUP
  C$$code > ROM
  Cidata > RAM
  Cudata > RAM
}
$icc$SP_VALUE = 0x203ffc;
```

Директива SECTIONS

Директива SECTIONS отображает область AREA, используемую при вводе в ассемблер на области памяти, перечисленные в директиве MEMORY:

```
SECTIONS
{
  <area name> : <MEMORY area name>
  ...
}
```

<area name> должно быть именем области во входных файлах ассемблера, а <MEMORY area name> должно быть одним из имен, определенных в директиве MEMORY.

Определение внешних символов

Вы можете определять значения глобальных символов в командном файле компоновщика:

```
<symbol> = <expression>;
```

Обратите внимание, что каждое выражение должно быть завершено точкой с запятой.

12.3.2. Символы, генерируемые средой разработки

Среда разработки передает компоновщику некоторые специфические для него параметры проекта через значения нескольких символов. Обычно они используются файлом запуска для инициализации среды Си. Их значения основаны на выборе, которые вы делаете на странице 4.13. Опции гомпилятора: Целевое естройство. См. типовой 7.2. Файл запуска об их использовании.

iccSP_VALUE

Вершина стека. Обычно это конец SRAM.

iccIRQ_VALUE

Размер стека IRQ.

iccFIQ_VALUE

Размер стека FIQ.

iccREMAP

Состояние флага “Memory REMAP”. Это может использоваться для условного разрешения контроллера памяти устройства и генерации команды REMAP.

12.3.3. Специальные символы компоновщика

Для поддержки среды Си, кроме 12.2.17. Предопределенные области ассемблера, поддерживаются также некоторые специальные символы. Их значения определяются компоновщиком:

iccROM_ADDR

Символ является адресом инициализированных значений объектов области Cidata. Вспомните, что Cidata размещается в оперативной памяти, поэтому фактические инициализированные значения должны быть сохранены в энергонезависимой памяти. Компоновщик помещает инициализированные значения в область C$$code после последней определяемой пользователем области C$$code. Следовательно, этот символ является суммой последней определяемой пользователем области C$$code и ее размера. 7.2. Файл запуска использует это, чтобы копировать значения в область Cidata.

iccRAM_ADDR

Адрес области Cidata. Файл Запуска копирует iccDATA_SIZE байтов из iccROM_ADDR в iccRAM_ADDR.

iccDATA_SIZE

Размер области Cidata.

iccURAM_ADDR

Адрес области Cudata. 7.2. Файл запуска обнуляет iccUDATA_SIZE байтов, начиная с iccURAM_ADDR.

iccUDATA_SIZE

Размер области Cudata. Пример его использования смотрите в исходном коде файла запуска CRT в директории c:\iccv7arm\libsrc.arm.

12.4. Библиотеки

Библиотека – это коллекция объектных файлов в специальной форме, которую понимает компоновщик. Когда объектный файл компонента библиотеки ссылается вашей программой непосредственно или косвенно, компоновщик извлекает из него библиотечный код и связывает его с вашей программой. Поддерживаемая стандартная библиотека – `libcarm.a`, которая содержит стандартные функции Си и функции, специфичные для ARM.Обратите внимание, что библиотечный файл должен иметь расширение `.a`. См. 12.3. Компоновщик. Имя программы библиотекаря – `ilibarm`.

12.4.1. Компиляция файла в библиотечный модуль

Каждый библиотечный модуль – это особая форма объектного файла. Следовательно, чтобы создать библиотечный модуль, вы должны скомпилировать исходный файл в объектный файл. Это может быть выполнено, открывая файл в среде разработки, и вызывая команду *File>CompileFileToObject*.

12.4.2. Распечатка содержания библиотеки

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

```
ilibarm -t libcarm.a
```

12.4.3. Добавление или замена модуля

Добавление или замена модуля:

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

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

```
cd \iccv7arm\libsrc.arm
<modify putchar() in putchar.c>
<compile putchar.c into putchar.o>
copy \iccv7arm\lib\libcarm.a ; copy library
ilibarm -a libcarm.a putchar.o
copy libcarm.a \iccv7arm\lib ; copy back
```

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