**Начальные сведения.**

**Как использовать компилятор С.**

**Orionsoft (C) 1996**

**Содержание.**

|  |  |
| --- | --- |
| **Начальные сведения о пакете BDS C** | **4** |
| **Простейший пример работы с пакетом** | **5** |
| **Cc - синтаксический анализатор** | **6** |
| **Примеры** | **8** |
| **Cс2 - генератор кодов.** | **8** |
| **Clink - компоновщик Си** | **9** |
| **Clib - библиотекарь Си** | **13** |
| **"Submit"-файлы ср/м** | **16** |
| **"Темнoe пятнo".** | **17** |
| **Формат "crl"-файлов и другие механизмы нижнего уровня** | **17** |
| **Формат crl в деталях** | **17** |
| **Оглавление crl-файла** | **17** |
| **Начало области внутренних данных и спецификации размера** | **18** |
| **Функциональные модули** | **18** |
| **Список необходимых функций** | **18** |
| **Длина тела функции** | **19** |
| **Тело функции** | **19** |
| **Параметры перемещения** | **19** |
| **Соглашения BDS C. Стек.** | **20** |
| **Указатель стека** | **20** |
| **Пространство, занимаемое стеком** | **20** |
| **Внешние данные** | **20** |
| **Протоколы вызова и возврата** | **21** |
| **Ввод/вывод файлов** | **22** |
| **Имена файлов** | **22** |
| **Префикс указателя диска** | **22** |
| **Префикс области пользователя** | **23** |
| **Обработка ошибок** | **23** |
| **Переполнение при записи** | **23** |
| **Первичные функции файлового ввода/вывода** | **23** |
| **Функции буферизованного ввода/вывода файлов** | **27** |
| **Особенности реализации** | **31** |
| **Комментарии** | **31** |
| **Идентификаторы** | **32** |
| **Ключевые слова** | **32** |
| **Примечания** | **32** |
| **Типичные ошибки** | **34** |
| **Индексация массива** | **35** |
| **Указатели** | **35** |
| **Возврат указателей** | **36** |
| **Формальные параметры** | **36** |
| **Вызовы функций** | **37** |
| **Оверлейная структура** | **38** |
| **Препроцессор ассемблера** | **41** |
| **Опции командной строки** | **41** |
| **Файловый ввод** | **43** |
| **Первичные функции файлового ввода/вывода** | **44** |
| **Функции буферизованного ввода/вывода файлов** | **48** |
| **Консольный ввод/вывод** | **52** |
| **Элементарный консольный интерфейс** | **52** |
| **BDOS все усложняет** | **53** |
| **Библиотека функций cio.crl** | **56** |
| **Пакет обработки чисел с плавающей точкой** | **56** |
| **Краткое описание функций** | **57** |
| **Общие вопросы** | **58** |
| **Длинные целые числа** | **59** |
| **Детали реализации** | **61** |
| **Отладчик для BDS C** | **61** |
| **Объяснение компонентов** | **62** |
| **Настройка и сборка отладчика** | **62** |
| **Куда поместить cdb2** | **62** |
| **Построение CDB** | **63** |
| **Построение cdb2** | **63** |
| **Изменение номера рестарта** | **65** |
| **Как активизировать отладчик** | **65** |
| **Активизация CDB** | **66** |
| **Резюме** | **66** |
| **Команды отладки** | **67** |
| **Точки останова** | **67** |
| **Выполнение программы** | **68** |
| **Дампирование переменных** | **69** |
| **Команды распечатки** | **70** |
| **Команда завершения** | **71** |
| **Алфавитный список команд отладки** | **71** |

**1. Начальные сведения о пакете BDS C.**

Основной пакет BDS C состоит из 4-х исполняемых файлов: cc.com - компилятор Си - фаза 1, cc2.com - компилятор Си – фаза 2, clink.com - компоновщик Си, clib.com - библиотекарь Си и трех служебных файлов, которые обычно требуются для компоновщика: c.ccc - инициатор выполнения и языковая среда (ниже - пакет времени выполнения), deff.crl -библиотека стандартных функций (по умолчанию), deff2.crl - библиотека дополнительных функций.

Cc.com и cc2.com вместе образуют компилятор, как таковой. Cc читает заданный исходный файл с диска, обрабатывает его и, оставив промежуточный файл в памяти, автоматически загружает cc2, чтобы завершить компиляцию и образовать файл типа ".crl".

Это означает, что промежуточный файл, созданный cc, может быть записан на диск и обработан cc2 отдельно. В этом случае промежуточный файл приобретет расширение ".cci".

Файл типа ".crl" (сокращение от "c relocatable») содержит сгенерированный машинный код 8080 в специальном перемещаемом формате.

Компоновщик (clink) принимает на вход "crl"-файл, содержащий функцию "Main" и переходит к поиску во всех заданных ему файлах типа ".crl" (а затем автоматически в deff.crl, deff2.crl и deff3.crl ) необходимых функций. Когда все такие функции подключены, создается "com"-файл.

Имеется также удобная программа clib для манипуляции содержимым файла типа ".crl".

Имеется несколько свойств, которые может задавать сам пользователь в cc.com и clink.com, изменяя содержимое особых байтов памяти, которые расположены в районе начала каждого исполняемого файла. Для изменения этих свойств при помощи ddt или sid считайте cc.com или clink.com в память, произведите изменения, используя команду s, нажмите control-c и посредством команды CP/M save запишите модифицированный исполняемый файл назад на диск.

Для преобразования, печатаемого ddt и sid шестнадцатеричного адреса "next" в десятичное число, которое вы должны задать команде save, чтобы сохранить модифицированную версию на диске, используйте следующий алгоритм: сначала возьмите две крайние слева цифры у шестнадцатеричного числа и вычислите их десятичный эквивалент (например, имея 3с80h, получим 3сh, равное десятичному 60). Затем нужно из результата вычесть 1, но только, если две крайние справа цифры являются нулями (например, выше указанные 60 останутся 60, так как правые крайние две цифры 3с80h - 80h, а не 00). Полученное в итоге значение - это и есть то число, которое должно быть задано в команде save.

И cc.com, и clink.com содержат одинаковый по структуре 5-и байтовый блок параметров. Базовый адрес блока для cc.com -0155h, а для clink.com - 0103h. структура блока следующая: база+0 - диск по умолчанию для поиска библиотеки 0ffh (текущий). База+1 область пользователя по умолчанию для поиска библиотеки - 0ffh (текущая). База+2 - диск, где обрабатываются файлы submit - 0 (диск "A"). База+3 - опрос консоли по прерываниям (0 или 1) 1 (разрешен). База+4 исполнять горячий старт по окончании - 0 (нет).

Следует отметить, что каждый параметр в блоке имеет размер ровно 1 байт. Первые два конфигурационные байта указывают диск и область пользователя, на которых cc и clink будут по умолчанию искать требуемые им библиотеки.

Для cc это оглавление определяет, где найти файлы, подключаемые директивой #include, когда имя файла заключено в угловые скобки, а также, где найти cc2.cом для второй фазы компиляции.

Для clink это означает, где найти файлы deff.crl, deff-2.crl и deff3.crl (если они есть) и c.ccc, а также, где получить другие crl-файлы, указанные в командной строке clink, если их нет в оглавлении, из которого был взят "главный" "crl"-файл.

Для "диска по умолчанию для поиска библиотеки" значение 0 специфицирует диск "A", 1 специфицирует диск "B" и т.д., a значение 0ffh (255 десятичное) специфицирует текущий диск по умолчанию. Для "области пользователя по умолчанию для поиска библиотеки" значения 0-15 означают соответствующую область пользователя, а значение 0ffh (255 десятичное) специфицирует, что по умолчанию должна быть взята область текущего пользователя.

Третий параметр указывает, какой диск CP/M содержит файл $$$.sub, который существует во время обработки "submit"-файлов. Возможные значения этого параметра такие же, что и параметра "диск по умолчанию для поиска библиотеки", описанного выше.

Clink всегда старается стереть этот файл, когда происходит ошибка, в то время как cc пытается это сделать только тогда, когда задана опция -х.

Так как большинство систем всегда помещает файл $$$.sub на диск "A", cc и clink настраиваются в дистрибутивной версии на этот диск. Но если пользователь привык, что его система вместо того, чтобы помещать файл $$$.sub всегда на диск "A", помещает его на текущий диск, то этот байт-параметр следует изменить с 0 на 0ffh.

Четвертый конфигурационный байт - это флажок, сообщающий cc или clink, будет или нет системная консоль реагировать на символ прерывания (control-c) во время выполнения команды.

Если реакция разрешена (не нуль), любой ввод с консоли пользователем во время выполнения команды, будет игнорироваться, за исключением control-c, в этом случае выполнение команды будет немедленно прервано и управление вернется на командный

уровень. Если запрещена (нуль), консоль никогда не будет опрашиваться. Это полезно в определенных системах, которые умеют воспринимать незапланированный ввод и не требуют от транзитных программ опроса ими консоли.

Пятый (и последний) байт определяет необходимость выполнения cc и clink "горячего старта" по окончании их работы или возврат прямо в CCP без какой-либо активизации диска. Дистрибутивная версия настроена на возврат прямо в CCP, но для некоторых систем класса CP/M (например, Cromix CP/M) прямой возврат в CCP работает неправильно.

Короче говоря, эта схема настройки позволяет пользователям, обладающим дисками больших объемов выбирать определенный диск и область пользователя, в которых хранятся все include-файлы и библиотечные файлы.

Байты библиотечного диска и области пользователя должны рассматриваться вместе, как одно целое. Если вы изменяете что-то одно, вы, вероятно, также захотите изменить и другое.

Следует отметить, что cc2.com нет необходимости перенастраивать: cc.com передаст ей всю нужную информацию при передаче управления.

**2. Простейший пример работы с пакетом.**

В качестве примера здесь представлена последовательность действий для компиляции и компоновки простого исходного файла с именем foo.c:

Компилятор вызывается командой:

A>cc foo.c <вк>

После печати сообщения о начале работы cc будет считывать файл foo.c с диска и обрабатывать его некоторое время. Если ошибок нет, cc выдаст диагностику использования памяти и загрузит cc2.com. Cc2 произведет дополнительную обработку и, если не произошло ошибки, запишет файл foo.crl на диск.

На следующем шаге вызывается компоновщик:

A>clink foo (другие файлы и опции, если необходимо) <вк>

Если нет неразрешенных ссылок, создается файл foo.com. Его можно выполнить при помощи команды:

A>foo (аргументы) <вк>

Внимание: командные строки для всех файлов типа ".com" в пакете должны вводиться в CP/M без пробелов в их начале (это также относится к "com"-файлам, сгенерированным компилятором), т.к. лидирующие пробелы в командной строке вызовут ошибки при определении ArgC и ArgV.

Ниже следует детальное описание синтаксиса команд.

**3. Cc - синтаксический анализатор.**

Формат команды: cc имя.расширение [oпции] <вк>

Допускаются любые имена и расширения, хотя обычное расширение для программ - это ".c". Cc прежде всего попытается открыть файл под явно заданным именем. Если расширение не указано и файл не может быть открыт в соответствии со спецификацией, cc добавит расширение ".c" к имени файла и попытается открыть его еще раз.

Если в имени файла указан диск (например "b:foo.c"), предполагается, что исходный файл находится на указанном диске, и вывод компилятора также будет направлен на этот диск. файлы в двойных кавычках в директиве "include" для которых отсутствуют спецификации области пользователя/диска, разыскиваются на том же самом диске, который извлечен из имени главного файла в командной строке.

Ввод символа control-c в любое время после вызова cc приведет к прекращению компиляции и возврату на командный уровень. Если cc не был на строен на игнорирование консоли, как описано выше.

Вслед за именем исходного файла может быть задан список опций компиляции, причем каждой опции предшествует тире. В настоящее время поддерживаются следующие опции: -р вызывает вывод исходного текста на консоль пользователя с автоматической генерацией номеров строк (после завершения всех операций #include и #define). Эта опция полезна для обнаружения неправильных ограничителей комментария: незакрытый комментарий приведет к тому, что часть текста не будет выведена, и выведенный фрагмент текста подскажет вам, где начинается неверно ограниченный комментарий.

Следует заметить, что этот вывод может быть направлен на системное устройство печати CP/M посредством выдачи control-p до вызова cc.

**-а d [n]** - автоматическая загрузка cc2.com из диска d, области пользователя n в случае успешного завершения работы cc.

По умолчанию cc2 должна размещаться или на текущем диске, или на диске/в области пользователя, подразумеваемых при настройке пакета.

Если при спецификации диска указана буква "z", то промежуточный файл с расширением ".cci" записывается на диск для последующей обработки при помощи явного вызова cc2, и не делается никаких попыток автоматически загрузить cc2.

**-d x** - приводит к тому, что выходной файл компилятора с расширением ".crl" записывается на диск x, если не произошло ошибок во время работы cc или cc2. Если задана также опция -а z, то -d специфицирует, на какой диск записывается файл cci. Диск назначения по умолчанию - это тот же самый диск, с которого был получен исходный файл.

**-м xххх** - специфицирует начальный адрес размещения пакета времени выполнения (c.ccc) в шестнадцатеричном виде. Применяется при использовании транслятора для генерации кода для нестандартной среды. По умолчанию пакет времени выполнения должен размещаться вначале TPA CP/M. Коды c.ccc, которые всегда размещаются в начале генерируемого файла типа ".cом", не могут быть отделены от функции Main и других (если они есть) функций корневого сегмента.

**-е хххх -** специфицирует в шестнадцатеричном виде точный адрес начала области внешних данных во время выполнения. Обычно внешние данные начинаются непосредственно за последним байтом программного кода и доступ ко всем внешним данным выполняется косвенно при помощи специального указателя, установленного clink в пакете времени выполнения. Когда используется -е, компилятор может при генерации кода использовать прямой доступ к внешним данным (используя lhld и shld), вместо того, чтобы использовать указатель на область внешних данных. Это сократит объем и увеличит эффективность программ с большим количеством внешних данных. Совет: не используйте эту опцию при отладке программы. Если программа уже отлажена, компилируйте ее с -е, поместив внешние данные на то же самое место, где они были прежде (в следующий раз код будет короче). Посмотрите на значение "последний кодовый адрес" из статистической распечатки clink, чтобы выяснить, на сколько сократился размер кода и затем компилируйте его весь опять, используя соответствующий новый адрес с опцией -е. Не располагайте его слишком близко, так как вы, вероятно, будете производить коррекцию в программе, что вызовет изменение ее длины и может привести к перекрытию специфицированной области внешних данных (эту ситуацию clink обнаружит и сообщит о ней). Опция -e будет иметь какой-то эффект только в случае успешной загрузки cc2 по окончании работы cc. Чтобы получить более детальное описание, см. также опцию -е компоновщика clink. Необходимо заметить, что clink будет печатать предупредительное сообщение, если адрес внешних данных, специфицированный этой опцией, попадает в область команд программы или операционной системы в генерируемом ".com"--файле .

**-о** - вызывает оптимизацию генерируемого кода по времени выполнения. Обычно при генерации кодов определенные сложные кодовые последовательности заменяются вызовами эквивалентных подпрограмм из пакета запуска, в то время, как это сокращает длину кода, это также замедляет выполнение из-за частого обращения к подпрограммам. Если используется -о, то многие подпрограммные вызовы замещаются встроенными кодами. В результате ускоряются, (но удлиняются) объектные программы. Для получения самого быстрого возможного кода должна использоваться опция -е совместно с -о. Для получения самого короткого возможного кода используйте -е, но не используйте -о. Опция -о будет иметь какой-то эффект только в случае успешной загрузки cc2 по окончании работы cc.

**-х** - вызывает уничтожение следов пакетной обработки CP/M ("submit"-файлов), в которых вызывался компилятор, если в процессе компиляции обнаружены какие-нибудь ошибки. Когда бы ни вызывался cc из файла submit, в командной строке должна появиться опция -х, если необходимо, чтобы временный файл $$$.sub до возвращения в командный уровень был удален после ошибочной компиляции. Когда cc используется самостоятельно, -х

вызовет только бесполезную активизацию дисковода.

**-r x** - резервирует х кбайт для таблицы символов. Если происходит ошибка "out of symbol table space" (выход за пределы таблицы символов), эта опция может использоваться, чтобы увеличить размер таблицы символов. Напротив, если вы получаете ошибку "out of memory" (нехватка памяти) можно использовать -r, чтобы уменьшить размер таблицы символов и обеспечить больше места для исходного текста. Но наилучшим выходом в случае нехватки памяти было бы разбить исходный файл на более мелкие части. Размер таблицы символов по умолчанию -10к.

**-c** - запрет вложенных комментариев (comment nesting). Обработка комментариев производится таким же способом, как и в Unix C, т.е. если задана -c, то такая строка, как

/\* PrintF("Hello !!!"); /\* this prints hello \*/

будет рассматриваться как завершенный комментарий. Если –с не используется, компилятор будет ожидать другую последовательность "\*/". Только после этого такой комментарий будет считаться завершенным.

Один исходный файл на Си не может содержать более 63 определений функций. Помните, что Си-программа может быть создана из любого числа исходных файлов, каждый из которых содержит до 63 функций.

Если при работе cc обнаружены ошибки, процесс компиляции будет тут же прерван вместо того, чтобы перейти ко второй фазе компиляции или записи файла .cci на диск (в зависимости от заданных опций).

Скорость обработки: около 20 строчек текста в секунду после того, как исходный файл загружен в память, не будет производиться никаких доступов к диску до завершения обработки. До тех пор, пока не истекли (n/20) секунд со времени последней активизации дисковода (где n - это число строк в исходном файле), не стоит волноваться за судьбу компиляции. По истечении этого времени можете беспокоиться.

**4. Примеры.**

A>cc foobar.c -r12 -ab <вк>

Запускает cc для компиляции файла foobar.c, при этом размер таблицы символов задается равным 12к. Cс2.com автоматически загружается с диска "B".

A>cc c:belle.c -p -o <вк>

Запускает cc для файла belle.c на диске "C". Текст файла после обработки инструкций #define и #include печатается на консоли (с номерами строк) .

Если cc не находит ошибок, cc2.com автоматически загружается или из текущего оглавления или из оглавления по умолчанию (оглавление - это пара диск-область пользователя). Производится оптимизация по времени.

**4. Cс2 - генератор кодов.**

Формат команды: cc2 имя <вк>

Обычно cc2.com загружается автоматически cc и эта команда не используется при явном вызове cc2, файл "имя.cci" будет загружен в память и обработан.

Если не произошло ни одной ошибки, выходной файл под именем "имя.crl" записывается на диск, а файл "имя .cci" (если имеется) удаляется.

Cс2 не имеет никаких опций.

Что касается cc, то явное указание диска в имени файла вызывает использование этого диска как для ввода, так и для вывода. Когда cc автоматически загружает cc2, несколько байтов в кодовой части cc2 устанавливаются в соответствии с опциями, заданными в командной строке cc. Если cc2 активизируется явным образом (т.е., не загружается автоматически), пользователь должен проследить, чтобы эти значения были установлены соответствующим образом до того, как начнется выполнение. Обычно этого не требуется, но, если у вас очень мало памяти на диске и необходимо активизировать cc2 отдельно, то некоторые значения установить необходимо.

Адрес Значение Опция Функция hex по умолчанию

0103 00 "-a" не ноль, если cc2 была автоматически загружена, иначе 0

0104 01 "-o" нуль,если задана -o (оптимизируется время выполнения), иначе 1

0105 ──

0106 01 "-m" исходный адрес с.ccс во время запуска

0107 ──

0108 ── "-е" явно указанный адрес области внешних данных

(если задана опция -е для cc)

0109 00 "-е" не ноль, если явно задан адрес области внешних данных, иначе 0

Байты 16-битовых значений должны быть заданы в обратном порядке (младший байт - первым, старший - последним). Скорость выполнения cc2: около 70 строчек в сек. (имеются в виду строки исходного текста).

Если control-c ввести с консоли в любой момент во время выполнения, компиляция будет прервана и управление вернется в командный уровень.

Пример:

A>cc2 foobar <вк>

**5. Clink - компоновщик Си.**

Формат команды: clink имя [другие имена и опции] <вк>

Файл имя.crl долен содержать функцию "Main".

Для файла "имя.crl" и всех других задаваемых crl-файлов (до появления опции -f) будут включены все их функции в загрузочный модуль. Если в командной строке задана опция -f, то все указываемые после нее файлы crl загружаются частично - только те функции, которые вызываются из уже загруженных функций (как из предыдущих файлов crl, так и из только что скопированного), включаются в исполняемый файл. После того, как поиск во всех перечисленных crl-файлах закончен, автоматически просматриваются стандартные библиотечные файлы deff\*.crl для поиска необходимых библиотечных функций. Порядок, согласно которому clink ищет библиотечные файлы, всегда одинаков: сначала deff.crl, затем deff2.crl и наконец, если это предусмотрено пользователем, deff3.crl.

Если программист использует функции, которые имеют те же имена, что и какие-либо функции из автоматически копируемых библиотечных файлов, такие функции должны быть помещены в один из crl-файлов, указанный явно в командной строке.

Функция, находящаяся в deff3.crl, не будет использована, если функции с совпадающими именами имеются в deff.crl и deff-2.crl (не были удалены из этих файлов).

По умолчанию clink принимает, что неявно заданные "crl"-файлы размещаются на текущем диске и что все библиотечные файлы (с.ccс и deff\*.crl) размещаются на диске и в области пользователя в соответствии со спецификацией в блоке настройки.

Если имя главного файла на командной строке содержит явное указание диска, данный диск становится диском по умолчанию для всех "crl"-файлов, указанных в командной строке. Каждый дополнительный "crl"-файл может иметь префикс диска в форме "d:" и/или префикс области пользователя в форме "nn/" чтобы специфицировать точное местонахождение файла. Если используются оба префикса, префикс области пользователя должен идти первым.

Если заданный "crl"-файл не может быть найден в соответствии с выше указанными правилами поиска, тогда просматривается оглавление, определяемое диском и областью пользователя по умолчанию, что позволяет пользователю размещать совместно используемые библиотечные файлы на одном диске/области пользователя по умолчанию и иметь свободный доступ к ним во время компоновки, выполняемой на различных дисках и в областях пользователя.

Если после просмотра всех заданных "crl"-файлов остаются какие-либо неразрешенные ссылки, clink перейдет в интерактивный режим. На экран будут выведены имена всех ненайденных функций и будет дана возможность указать для поиска другие файлы.

Во время выполнения может быть введен символ control-c для того, чтобы прекратить компоновку и вернуться на командный уровень.

Вперемешку с именами файлов в списке, которые нужно искать, могут располагаться определенные опции редактирования, которым предшествует тире. Необходимо отметить, что несколько опций могут быть объединены, следуя за одним тире.

Ниже представлены реализованные опции:

"-s" - распечатать карту распределения памяти и краткую статистику на консоли.

"-f имя..." - вызывает выборочное включение функций изо всех файлов, расположенных вслед за ней в командной строке, вместо обычного включения этих файлов целиком.

Clink обычно загружает все функции из каждого "crl"-файла, указанного в командной строке, до тех пор, пока не встретится эта опция. После "-f" в программу включаются только те функции, на которые имелись ссылки в текущем файле или в одном из предыдущих файлов.

"-е хххх" - предписывает расположить область внешних данных с адреса хххх (шестнадцатиричное). Обычно область внешних данных следует непосредственно после сгенерированного кода, однако можно использовать эту опцию для того, чтобы отвергнуть это умолчание. Это необходимо, когда выполняется подцепление (посредством Exec, ExecL или ExecV), чтобы быть уверенным в том, что представление новой программы о том, где начинаются внешние данные то же самое, что и у старой программы. Чтобы выяснить, какое использовать значение, нужно прежде всего выдать команду clink на все "crl"-файлы, с опцией "-s", но без опции "-е", отмечая адрес "data starts at:" ("данные начинаются с"), выданный clink для каждого файла. Затем от компонуйте их опять, используя максимум всех тех адресов в качестве xxxx после "-е" для всех файлов за исключением того, который имеет наибольший адрес начала данных во время первого прогона.

При генерации кода для Rom эта опция должна использоваться, чтобы разместить внешние данные по определенному адресу в памяти.

Если "crl"-файл был получен в результате компиляции с опцией "-е", заданной в командной строке cc, то clink автоматически узнает адрес, который был специфицирован в командной строке cc. Но если любой из других заданных "crl"-файлов в командной строке clink содержит функции, компилированные cc без использования опции "-е" или со значением, заданным "-е", которое отличается от значения, использованного для компиляции главной функции, полученный в результате "com"-файл не будет правильно работать. "Crl"-файлы, полученные без использования опции "-е" cc могут включаться в "com"-файл, только если "-е" задано для clink с аргументом, точно равным аргументу опции cc "-е", использованному для компиляции основного "crl"-файла. "-z" - запрещает очистку области внешних данных (заполнение нулями) во время запуска. Если используется "-z", все внешние данные имеют случайные значения.

"-t хххх" - устанавливает начальный адрес стековой памяти равным xххх (шестнадцатиричное). В начало сгенерированного кода помещается команда "lxi sp, xxxx". Обычно, т.е. если нет опции "-t", сгенерированный "com"-файл начинается с последовательности команд: "lhld base+6", чтобы получить указатель bdos из базовой страницы. "Sphl" - инициировать указатель стека значением базы. Значение должно быть достаточно большим, чтобы позволить всему программному коду и локальным/внешним данным разместиться в памяти во время загрузки. Если Вы генерируете код для прогона в Rom, значение, заданное здесь, должно быть равно максимальному адресу плюс единица в памяти чтение/запись, которая должна использоваться для стека.

"-o имя" - присваивает выходному файлу com имя "имя.com". Если перед именем стоит указатель диска, то вывод записывается в этот диск. По умолчанию вывод идет в текущий зарегистрированный диск. Если записан указатель диска в виде одной буквы, за которой следует двоеточие, без имени файла, то "com"-файл направляется на специфицированный диск под тем же именем, что и "crl"-файл.

"-n". При задании этой опции полученный в результате "com"-файл сохраняет CCP (процеccор консольных команд) CP/M в памяти во время выполнения, вместо перекрытия его стеком времени выполнения. Это сокращает доступную исполняющемуся файлу память на 2 кбайта, но позволяет программе моментально вернуться на командный уровень после ее выполнения без горячего старта с диска. Поэтому опция "-n" полезна для программ, которые используются часто и не слишком стеснены нехваткой памяти.

"-n" игнорируется, если одновременно используется опция "-t", так как эти средства конфликтны и "-t" дан больший приоритет.

"-w" - записывает файл таблицы символов с именем имя.sym на диск, где "имя" то же самое, что и у полученного в результате "com"-файла. Этот файл содержит имена и абсолютные адреса всех функций, включенных в загружаемый модуль. Он может использоваться с sid с целью отладки или совместно с опцией "-y", при создании оверлейной структуры.

"-y имя" - считывает файл с таблицей символов, под именем "имя.sym" из диска и использует адреса всех точек входа, определенных в нем, для текущей компоновки.

Опции "-w" и "-y" созданы для совместного использования при создании оверлейной структуры выполняемой программы. Например: при компоновке корневого сегмента (той части программы, которая загружается с начала TPA CP/M, первой получает управление и содержит языковую среду), необходимо задать опцию "-w", чтобы записать файл таблицы символов, содержащий адреса функций, находящихся в корневом сегменте. Затем, при компоновке оверлейных сегментов используется опция "-y", чтобы прочитать таблицу символов родительского сегмента и, таким образом, предотвратить присутствие множества копий общих библиотечных функций во время запуска. Эта процедура может быть продолжена на несколько уровней: при редактировании связей для оверлейного сегмента можно указать опцию "-w" совместно с опцией "-y", что приведет к созданию расширенного файла символов, содержащего помимо символов из считанного по "-y" файла и вновь определенные локальные символы. Оверлейный сегмент может иметь подчиненные ему оверлейные сегменты и т. д. на столько уровней, на сколько хочется (или нужно). Необходимо отметить, что расположение опции "-y" в командной строке clink очень важно, т.е. поиск файла символов, указанного в опции, начнется только после того, как все заданные слева от "-y" уже использованы clink. Таким образом, лучше указать "-y" сразу после имени главного "crl"-файла. Если после чтения символов из sym-файла обнаружено, что некоторый символ имеет то же самое имя, что и уже определенный символ, печатается сообщение об этом на консоли и сохраняется старое значение символа. Чтобы получить более подробную информацию об использовании "-y" для генерации оверлейной структуры, см. соответствующее приложение.

"-l хххх" - устанавливает адрес загрузки сгенерированного кода равным хххх (шестнадцатеричное). Эта опция необходима тогда, когда генерируется оверлейный сегмент (вместе с "-v").

"-c d [n]" - предписывает clink получить библиотечные файлы (deff.crl, deff2.crl, c.ccc и, возможно, deff3.crl) и любые файлы crl, заданные в командной строке, но не найденные в текущем оглавлении (или на диске, специфицированном как префикс к имени главного "crl"-файла), из диска d и области пользователя n.

Эта опция используется, чтобы отвергнуть спецификацию диск/область пользователя по умолчанию, постоянно "запаянную" в блок настройки clink. Библиотечный файл deff3.com, создаваемый пользователем, автоматически сканируется, если этот файл существует и еще имеются неразрешенные ссылки после того, как были сканированы deff.crl и deff2.crl. Если deff3.crl не найден, редактор ничего не предпринимает.

"-d ["aрг"]" - для быстрого тестирования. -d вызывает немедленное выполнение "com"-файла вместо записи его на диск. Если после "-d" в кавычках задана строка аргументов, эффект будет таким, как если бы программа была вызвана из CCP с параметрами из этой строки. "-d" не должна использоваться для сегментов, у которых адреса загрузки не совпадают с началом области TPA (т.е. "-d" должна быть использована только для корневых сегментов.

"-d" будет проигнорирована, если задана опция "-n".

"-r" xxxx резервирует хххх (шестнадцатеричное) байтов для таблицы ссылок вперед (по умолчанию до 600h). Эта опция может использоваться, чтобы распределить больше памяти, при возникновении ошибки "ref table overflow" (переполнение таблицы ссылок).

Примеры:

A>clink ted -s -t 600 -o joyce <cv>

Здесь clink ожидает, что файл ted.crl содержит функцию "Main", к которой затем подключаются функции из ted.crl и другие необходимые функции из deff.crl, deff2.crl и, если существует, deff3.crl. По окончании работы печатается статистика. Начало программного стека устанавливается в 6000h (оставляя память 6000н и выше нетронутой "com"-файлом при выполнении).

Результат компоновки получает имя joyce.com.

A>clink b:lois 6/c:vicky -f janet -s <вк>

В этом примере clink загружает все функции из lois.crl (из диска "B") и vicky.crl (в области пользователя 6 на диске "C"), подключает необходимые функции из janet.crl (из диска в, так как диск, откуда был извлечен lois.crl, является умолчанием для этого редактирования), deff.crl, deff2.crl и, может быть, deff-3.crl и, закончив работу, печатает краткую статистику. Так как не указана опция -t, clink считает, что под данные и команды можно отводить всю ОТП (область транзитных программ). Генерируемый "com"-файл именуется lois.com по умолчанию (т.к. нет опции -o) и записывается на текущий зарегистрированный диск.

Примечание: когда несколько файлов, которые совместно пользуются внешними переменными, скомпонованы вместе, файл, содержащий функцию "Main" должен иметь объявления всех этих данных на внешнем уровне (часто это делается посредством #include-файлов, которые пишутся авторами программ-источников вешних данных). Если этого не сделано возможно перекрытие внешних данных.

**6. Clib - библиотекарь Си.**

Формат команды: clib <вк>

Программа clib позволит Вам:

a) перемещать функции из одного "crl"-файла в другой;

b) переименовывать, удалять и проверять наличие отдельных функций;

c) создавать новые "crl"-файлы;

d) проверять содержимое "crl"-файла.

Прежде, чем детально изучить работу clib, полезно понять структуру "crl"-файлов.

"Crl"-файл состоит из набора независимо откомпилированных Си-функций, каждая - образ двоичного машинного кода 8080 с начальным адресом, равным нулю. Чтобы реализовать перемещающиеся адреса, clink использует список "параметров перемещения", который хранится в теле каждой функции. Там же хранятся имена всех подчиненных функций, вызываемых данной функцией. Код, список перемещения и список необходимых функций называются функциональным модулем.

Первые четыре сектора "crl"-файла составляют оглавление для этого файла, содержащее список всех функциональных модулей, появляющихся в файле, и их расположение в файле. Общий размер "crl"-файла не может превосходить 64 кбайт (т.к. функциональные

модули размещаются через двухбайтовые адреса), но оптимальная эффективность достигается при сокращении размера "crl"-файла до величины одного экстента CP/M (16 кбайт).

После запуска clib появится начальное сообщение и будет указан размер буфера для функций. Размер буфера говорит о том, сколько имеется памяти для размещения функций во время работы с ними. Попытки переместить или извлечь более длинные функции будут безуспешными.

Вслед за инициализацией clib выдаст подсказку (звёздочка) и будет ждать команду.

Чтобы открыть "crl"-файл для работы с ним, введите:

\*оpen <номер файла> [d:] <имя файла> <вк>,

где <номер файла> является одноцифровым идентификатором (0-9), специфицирующим, какой "номер файла" должен ассоциироваться с файлом <имя файла> все то время, пока этот файл остается открытым. Следовательно, одновременно может быть открыто до десяти файлов.

Следует заметить, что в имени файла может быть специфицирован указатель диска, что позволяет работать с "crl"-файлами на любом физическом диске.

Чтобы закрыть файл (закрепляя все изменения, которые в нем были сделаны), необходимо набрать команду:

\*close <номер файла> <вк>

Тогда номер данного файла становится свободным и его можно приписать новому файлу при открытии. Следует отметить, что на операцию close может потребоваться определенное время.

Нет необходимости закрывать файл, если в нем не было сделано никаких изменений, или вам не требуется дополнительный номер файла. Например, файл, открытый только для копирования, не нужно закрывать.

Когда "crl"-файл открыт, копия оглавления файла (первые 4 сектора) загружается в память. Любые изменения файла (путем использования команд добавления, перемещения, переименования и/или стирания) вызывают соответствующую модификацию оглавления в памяти, но для фиксирования этих изменений на диске необходимо закрыть файл. Таким образом, если вы хотите отказаться от изменений в случае неуверенности в их правильности, и вы еще не закрыли файл, вы можете сделать недействительными все изменения файла, просто не закрыв его. Чтобы ликвидировать произведенные добавления и перемещения, необходимо немного дополнительной работы; это будет объяснено позже.

Чтобы увидеть список всех открытых файлов вместе с соответствующей статистикой, введите :

\*files <вк>

Чтобы просмотреть содержимое определённого "crl"-файла и увидеть размер входящих в него функций, введите

\*list <номер файла> <вк>

Имеется несколько способов перемещения функций между "crl"-файлами. Когда все необходимые файлы открыты, самый простой способ перезаписи функции (или набора функции) - это

\*transfer <номер-1> <номер-2> <имя функции> <вк>

Команда переписывает указанную функцию (или несколько функций) из файла, открытого под номером <номер-1> в файл, открытый под номером <номер-2>, не стирая ee из файла-источника.

<Имя функции> может включать специальные знаки \* и ?, если необходимо неявное имя. Будут переданы все функции, подходящие для неявного имени.

Альтернативный подход - использовать метод "извлечение-добавление". Команда извлечения имеет вид:

\*extract <номер файла> <имя функции> <вк>

При этом функция извлекается из данного файла и помещается в функциональный буфер (в память). Чтобы записать функцию в другой файл, введите:

\*аppend <номер файла> [<имя>] <вк>

где имя является необязательным и требуется только в случае, если нужно изменить имя, под которым должна быть сохранена функция:

\*аppend <номер файла> <вк>

достаточно для записи функции в файл без изменения ее имени.

Только один файл может быть задан в команде append. Чтобы записать функцию в несколько "crl"-файлов, для каждого файла должна быть выполнена отдельная команда append. Чтобы переименовать функцию в пределах определенного "crl"-файла, введите:

\*rename <номер файла> <старое имя> <новое имя> <вк>

Следует заметить, что это означает изменение в файле и нужно закрыть его, чтобы зафиксировать изменение.

Чтобы создать новый пустой "crl"-файл, введите

\*make <имя файла> <вк>

При этом создается файл на диске, под именем <имя файла>.crl; его содержимое вначале пусто. Чтобы записать в него функции, сначала откройте его, затем используйте метод передачи или метод извлечения/добавления, описанные выше.

Clib не разрешит создание нового "crl"-файла с тем же самым именем что и существующий "crl"-файл в том же оглавлении.

Чтобы удалить функцию (или набор функций) из файла, используйте

\*delete <номер файла> <имя функции> <вк>

Опять-таки, <имя функции> может быть задано неявно, т.е. содержать знаки \* и ?. Естественно, файл должен быть закрыт, чтобы удаление действительно было выполнено. Следует заметить, что удаление функции не освобождает элементов оглавления в соответствующем "crl"-файле до тех пор, пока этот файл действительно не закрыт. Таким образом, если оглавление "crl"-файла переполнено, и вы желаете поместить в него еще несколько функций, прежде всего следует удалить ненужные функции, затем закрыть и еще раз открыть файл, чтобы передать в него новые функции.

Краткий перечень команд может быть выдан на экран, если ввести команду:

\*help <вк> или \*?

Чтобы выйти из clib и вернуться на командный уровень, наберите команду:

\*quit <вк>

и дайте положительный ответ на запрос, выданный clib.

Примечание: все команды clib могут быть сокращены до одной буквы. Если вы вдруг решили, что вы на самом деле не хотели делать тех манипуляций с файлом, которые были произведены вами, может быть использована команда quit, чтобы избежать порчи файла и ликвидировать все его изменения.

Если вы еще не применяли к файлу команд append или transfer, введите:

\*quit <номер файла> <вк>

Этого достаточно, чтобы отказаться от всех изменений данного файла и освободить <номер файла>, как будто бы он был закрыт.

Если вы уже добавили или переместили функцию в файл и хотите отказаться от этого, тогда должна опять-таки быть использована команда quit, но кроме этого, нужно еще раз открыть файл прямо после quit и затем немедленно закрыть его. Эти махинации объясняются следующим образом: когда вы делаете добавление или передачу, добавленная функция записывается в конец "crl"-файла. Когда вы отказываетесь от добавления, в оглавление файла записывается прежнее его содержимое (до изменения), но добавленная функция все еще на диске, она занимает дисковую память, даже если она отсутствует в оглавлении. Открывая и тут же закрывая файл, вы оставите в нем только те функции, которые перечислены в оглавлении, успешно избавившись от "призрачных" функций.

Ниже дан пример сеанса работы с clib, в котором пользователь создает новый "crl"-файл под именем new.crl на диске "B", содержащий все функции из deff.crl, имена которых начинаются с буквы "р":

A>clib

BD Sofware C librarian v1.50

function buffer size = xxxxx bytes

\*open 0 deff

\*make b:new

\*open 1 b:new

\*transfer 0 1 p\*

\*close 1

\*quit

(quit) are you sure ? y

А>

**7. "Submit"-файлы ср/м.**

Чтобы упростить процесс компиляции и компоновки программы на "Си" (после того, как первоначальные ошибки удалены и Вы чувствуете себя уверенными в том, что сс и сс2 не найдут новых ошибок в исходном файле), нетрудно создать submit-файлы CP/M, чтобы выполнить всю работу. Самая простая форма файла submit, для простой компиляции, компоновки и исполнения исходной программы на Си, которая не требует других "crl"-файлов для сцепления), выглядит, к примеру, так:

cc $1.c

clink $1 -s

$1

Так что если вы хотите компилировать исходный файл под именем, скажем, life.c, введите только:

A>submit c life<вк>

(предполагается, что submit файл именуется c.sub).

**8. "Тёмное пятно".**

Когда активизируется какой-то файл пакета BDS C или "com"-файл, сгенерированный компилятором, передаваемая ему командная строка CP/M не должна содержать впереди себя пробелов или знаков табуляции. Похоже, CCP (процессор консольных команд) не анализирует командную строку должным образом, если в ее начале имеются пробелы.

**9. Формат "crl"-файлов и другие механизмы нижнего уровня.**

Эта глава адресована программистам, пишущим программы на языке ассемблера (или в машинном коде), которым необходимо связывать подпрограммы в машинных кодах с программами, полученными в результате работы компилятора "Си". В данной главе детально описан формат crl и описано, как получать функции на языке ассемблера в формате crl таким образом, чтобы их можно было использовать, как любые другие функции, компоновщиком "Си". Описаны соглашения по передаче параметров и вызову, используемые в "Си", а также некоторые полезные подпрограммы, имеющиеся в пакете времени выполнения.

**10. Формат crl в деталях.**

Для использования ассемблера фирмы digital research - asm на стандартном дистрибутивном диске имеется программа под названием casm.com. Эта программа принимает на вход функции на языке ассемблера в специальном формате "csm". Затем автоматически преобразует исходный файл типа csm в файл типа asm, пригодный для ассемблирования посредством asm.com.

Хотя для эффективного использования casm и asm в целях создания crl файлов нет никакой необходимости знать, как организован crl-файл, для завершенности изложения ниже дается детальное описание формата crl.

**11. Оглавление crl-файла.**

Первые четыре сектора "crl"-файла занимает его оглавление. Если вы используете ddt или sid для просмотра файла, эти секторы расположены в ячейках памяти 100h - 2ffh.

Каждый функциональный модуль в файле имеет соответствующий вход в оглавление, состоящий из имени модуля (не более восьми символов, причем старший бит в последнем байте установлен в 1) и двухбайтового значения, обозначающего адрес первого байта модуля в пределах файла. Примечание: адреса функциональных модулей внутри "crl"-файла соотнесены с позицией 0, то есть взяты относительно начала файла. При этом оглавлению соответствуют адреса с 0 по 1ffh. Ячейки 200h-204h зарезервированы, так что наименьший адрес функционального модуля равен 205h.

За последним входом в оглавление должно следовать пустое имя (80h), за которым следует слово, обозначающее адрес первого свободного байта в файле. Дополнение пробелами допускается после конца любого физического функционального модуля, чтобы установить адрес следующего модуля на определенную границу (скажем, 16 байт), но в самом оглавлении никогда не должно быть никакого дополнения пробелами.

**12. Начало области внутренних данных и спецификации размера.**

Первые пять байтов пятого сектора "crl"-файла (ячейки 200h-204h относительно начала файла) содержат информацию, которую clink использует, чтобы определить начало (если при запуске cc явно была задана опция -e) и размер области внешних данных для выполняющейся программы. Эта информация достоверна только, если "crl"-файл, содержащий ее, рассматривается как "главный" "crl"-файл в командной строке clink; в противном случае эта информация не используется.

Первый байт пятого сектора имеет значение 0bdh, если опция -е использовалась во время компиляции для создания области внешних данных; в противном случае его значением должен быть 0.

Второй и третий байты содержат адрес, заданный как операнд в опции -е, если она использовалась.

Четвертый и пятый байты пятого сектора содержат размер области внешних данных (первым - младший байт, вторым – старший байт). Clink всегда получает размер области внешних данных из этих специальных ячеек "основного" "crl"-файла (т.е. файла crl, содержащего функцию "Main" программы). В "crl"-файлах, кoтoрые не содержат функцию "Main", эти байты не используются.

**13. Функциональные модули.**

Каждый функциональный модуль в пределах "crl"-файла - это независимая единица, содержащая образ самой функции в двоичных кодах, плюс набор параметров перемещения для функции и список имен всех функций, которые она может вызвать.

Функциональный модуль - независим от адреса; это означает, что он может быть физически передвинут в любую ячейку в пределах файла crl (как это часто бывает, когда при использовании clib перемешиваются модули).

Формат функционального модуля:

1. список нужных функций

2. длина тела

3. тело

4. параметры перемещения

**14. Список необходимых функций.**

Если функция, которую вы создаете, вызывает другие функции, первым элементом в модуле должен быть список имен этих функций. Формат его следующий: это просто последовательный список имен с установленным в единицу старшим (седьмым) битом в

последнем байте каждого имени. Нулевой байт завершает список. Пустой список (когда функция не вызывает никаких других функций) - это только один нулевой байт.

Например, пусть функция foobar вызывает функции putchar, getchar, setmem.

Список функций, необходимых foobar, имеет следующий вид:

47 45 54 43 48 41 d2 50 55 54 43 48 41 d2

g e t c h a r' p u t c h a r'

53 45 54 4d 45 cd 00

s e t m e m' (конец)

**15. Длина тела функции.**

Далее идет 2х-байтовое слово, специфицирующее точную длину (в байтах) тела, оно должно следовать сразу за списком функций. Слово, определяющее длину, записывается следующим образом: сначала - младший байт, затем - старший байт.

**16. Тело функции.**

Этот раздел функционального модуля содержит машинный код 8080 для функции, причем стартовый адрес всегда равен нулю.

Если список необходимых функций был пустым, машинный код начинается с первого байта тела. Если список необходимых функций специфицировал n имен, в начале тела должна быть размещена таблица ложных векторов переходов, состоящая из n команд jmp), ей должна предшествовать команда перехода за эту таблицу.

Например, начало тела для функции foobar, описанной выше, будет таким:

jmp 000ch

jmp 0000h

jmp 0000h

jmp 0000h

**17. Параметры перемещения.**

Прямо вслед за телом располагаются параметры перемещения, т.е. набор адресов (относительно начала тела функции), указывающих на поля операндов каждой команды в пределах тела, ссылающейся на локальный адрес. Clink берет каждое слово, на которое указывал указатель в этом списке и прибавляет к нему базовый адрес времени выполнения.

Первое слово в списке перемещений - это просто длина списка параметров перемещения, таким образом, если имеется n параметров перемещения, длина списка (включая и слово, содержащее длину) будет 2n + 2 байтов.

Например, функция, которая содержит четыре локальных команды перехода, чьи коды операции размещены соответственно в ячейках 22h, 34h, 4fh, 61h, будет иметь следующий список перемещений:

04 00 23 00 35 00 50 00 62 00

Примечание: адреса команд должны быть увеличены на единицу, чтобы указать на действительные адреса перемещаемых операндов.

**Соглашения BDS C.**

**18. Стек.**

Передача всех аргументов вызываемой функции, так же, как и размещение (автоматическое) всей локальной памяти, происходит в одном стеке.

**19. Указатель стека.**

Указатель на стек хранится в регистре sp и инициируется значением адреса старшего байта памяти, доступной пользователю во время выполнения программы.

Предполагаемый последний адрес имеющейся памяти зависит от того, какие опции заданы во время компоновки; по умолчанию указатель стека инициализируется базовым адресом BDOS CP/M, и стек растет вниз, стирая CCP и требуя горячего старта после выполнения программы, чтобы возвратить CCP обратно в память. Если используется опция -t, то значение, заданное в качестве аргумента -t, используется для инициализации sp. Если использовалась опция -n, sp устанавливается на начало CCP, что уменьшает пространство стека на 2к, но позволяет возврат на командный уровень без выполнения горячего старта.

**20. Пространство, занимаемое стеком.**

Схема с одним стеком располагает всю локальную память данных (автоматическую), формальные параметры, адреса возврата и промежуточные результаты в одном стеке, который начинается в старших адресах памяти и растет вниз.

Максимальное пространство, требующееся для стека, примерно равно объему локальной памяти данных, распределяемой в случае наибольшей глубины вложенности вызовов функции, плюс примерно несколько сотен байтов (в наихудшем случае) для результатов промежуточных вычислений.

Если объем локальной памяти составляет n, то объем свободной памяти, доступной пользователю, можно выразить формулой:

topofmem () - endext () - (n + fudge), где значение fudge

(память под промежуточные результаты) около 500 является достаточно надежным.

Topofmem и endext являются библиотечными функциями, которые возвращают соответственно указатель на старший байт памяти, используемой работающей программой (вершина стека) и указатель на байт, который следует за концом области внешних данных. Значение еndext() - это указатель на первый байт памяти, имеющейся в распоряжении программиста для распределения и/или для общецелевого использования.

**21. Внешние данные.**

Внешняя память обычно располагается сразу после программного кода, при этом вся память между концом внешних данных и стеком свободна для распределения.

**22. Протоколы вызова и возврата.**

Когда функция получает управление, она обычно выполняет следующие действия в следующем порядке: распределить пространство для локальных данных в стеке (уменьшить SP на количество необходимой локальной памяти) и записать новое значение SP в регистр BC для использования в качестве постоянного указателя на основание блока данных. Причина записи SP в BC вместо обычной адресации через SP заключается в том, что SP сильно изменяется по мере поступления команд push и pop, делая подсчет различных адресов довольно запутанным. Следует заметить, что старое значение BC должно всегда быть сохранено для вызывающей программы. Допустим, что вызванная функция требует nlocl байт локального стекового пространства. После сохранения (push) старого значения BC, уменьшения SP на величину nlocl и записи SP в BC (именно в этом порядке), адрес любой локальной переменной со смещением loffset может быть просто подсчитан при помощи формулы:

(BC)+loffset

Если функции передаются параметры, адрес n-го фактического параметра может быть получен следующим образом:

(BC)+nlocl+2+2n,

где n равно 1 для первого параметра, 2-для второго и т.д. эта последняя формула получена из тех соображений, что параметры всегда записываются в стек в обратном порядке при вызове программы, и что запись аргументов является последней операцией, выполненной вызывающей программой перед непосредственным вызовом функции. После того, как вызванная функция сохраняет регистр bc, появляются 4 дополнительных байта в стеке между текущим значением указателя и первым формальным параметром:

1) сохраненный регистр bc,

2) адрес возврата в вызывающую функцию.

Следует заметить, что эта схема требует, чтобы каждый параметр занимал точно 2 байта. Поэтому параметры размером в один байт (типа char) всегда преобразуются в 16-разрядные значения при помощи добавления нулевого старшего байта без расширения знакового бита до их передачи в качестве параметров.

Завершив действия по обработке информации, вызванная функция перед возвратом управления должна освободить свою локальную память, увеличив sp на величину nlocl, восстановить регистровую пару bc, вытолкнув сохраненное значение из стека, затем она передает управление вызывающей программе.

Вызывающая программа несет ответственность за восстановление sp в прежнее значение, которое он имел до помещения в стек параметров вызова. Вызванная функция не может этого сделать, т.к. она не в состоянии определить действительное количество переданных параметров (которое может быть переменным, как в случае функции printf). Итак, формальные обязанности вызывающей функции:

1. Записать фактические параметры в стек в обратном порядке (последний параметр - первым, первый - последним);

2. Вызвать требуемую функцию, убедившись, что ни в регистре hl, ни в регистре de нет каких-либо важных значений (т.к. вызываемая функция может запортить содержимое регистров de, а в регистре hl - возвратить значение). При этом можно считать, что в результате вызова регистр bc не портится, т.к. по соглашению, вызываемая функция дожна сохранять содержимое регистров bc. Так поступают как функции, генерируемые компилятором, так и все ассемблерные функции пакета BDS C.

3. После возврата из вызванной функции: восстановить прежнее значение регистра sp, которое он имел до записи в стек формальных параметров. Будьте осторожны не - повредите при этом регистров hl, содержащих возвращенное функцией значение. Самый простой способ реставрации указателя стека - выполнить pop d для каждого занесенного в стек параметра. Обязанности вызываемой функции:

1. Записать в стек содержимое регистров bc, если имеется вероятность их изменения до возврата в вызывающую функцию.

2. Если требуется локальная память, уменьшить указатель стека на величину необходимого количества байтов.

3. При необходимости, сохранить новое содержимое sp в bc, чтобы использовать его в качестве указателя на базу блока локальных переменных. Не делайте этого, если вы не сохранили bc на шаге 1.

4. Выполнить требуемые вычисления.

5. По окончании вычислений освободить локальную память, увеличив sp на размер блока локальной памяти.

6. Извлечь старое значение bc из стека, если оно было сохранено на шаге 1.

7. Выполнить возврат вызывавшую программу, поместив возвращаемое значение в регистры hl, если оно имеется.

**23. Ввод/вывод файлов.**

В пакете BDS C имеется две основные категории функций файлового ввода/вывода. Первичные функции (нижнего уровня) используются для чтения данных с диска и записи на диск секторами. Буферизованные функции ввода/вывода позволяют пользователю иметь дело с данными в более удобных количествах, таких, как один байт или одна строка текста. Вначале опишем первичные функции, а затем буферизованные.

**24. Имена файлов.**

Если функция получает имя файла в качестве аргумента, это имя должно быть или константой-строкой символов, или любым выражением со значением, указывающим на строку, содержащую имя файла. Допустимы имена файлов, содержащие символы как верхнего, так и нижнего регистра, но внутри строки не должно быть символов пустого промежутка (возврата каретки, перевода строки, пробела, табуляции).

**25. Префикс указателя диска.**

Имя файла может содержать указатель диска в форме "d:", чтобы специфицировать определенный дисковод CP/M; по умолчанию это текущий диск. Символ d может быть любой буквой от а до z (соответствующей реальному логическому устройству).

**26. Префикс области пользователя.**

Спецификатор области пользователя в виде "#/" может также использоваться в качестве префикса имени файла, где # является десятичным числом от 0 до 31. Если префикс пропущен, по умолчанию принимается область текущего пользователя. Если заданы оба префикса (обозначение диска и спецификатор области пользователя), тогда префикс области пользователя должен быть первым. Например, чтобы открыть файл, под именем "foobar.zot" в области пользователя 7 на диске с, вы должны указать:

Open ("7/c: foobar.zot", Mode)

Если в имени файла будут обнаружены какие-либо непечатаемые символы (такие, как control-знаки), имя файла будет отвергнуто и функция возвратит значение Error. Это как-то облегчит трудности, вызванные попыткой открыть файл, имя которого содержит непечатаемые символы, но этот механизм все еще не является надежным. Будьте осторожны при построении имен файлов.

**27. Обработка ошибок.**

BDS C v.1.50 содержит новые средства диагностики ошибок ввода/вывода файлов. Обычно все стандартные функции в случае ошибки возвращают значение (-1) (Error). В таких ситуациях можно обратиться к функции ErrNo, чтобы получить специальный код ошибки, дающий более детальную информацию о ней. Если вы передадите значение, возвращенное функцией ErrNo, функции ErrMsg, ErrMsg возвратит указатель на строку, которая содержит словесное описание произошедшей ошибки. Здесь представлен пример использования этого механизма, в данном случае диагностировались ошибки, которые произошли при выполнении функции Write:

if(Write (FD, Buffer, NSects)= NSects )

PrintF("Write error: %s\n", ErrMsg (ErrNo())),

Следует заметить, что функция Write составляет исключение из правила, гласящего, что сигналом ошибки является возврат значения -1 (Error); Write возвращает число фактически записанных секторов; если это число не равно числу секторов, которые требовалось записать, очевидно, произошла ошибка.

**28. Переполнение при записи.**

Для обнаружения переполнения в большом файле, открытом для чтения/записи, имеется функция OFlow. Переполнение происходит, если вы пытаетесь читать/записывать после сектора номер 65535 файла.

**29. Первичные функции файлового ввода/вывода.**

**int Open (Filename,Mode)**

**char \*Filename;**

**char Mode;**

Открывает специфицированный файл для ввода, если Mode является нулем; вывода, если Mode равно 1; или ввода и вывода, если Mode равно 2.

Возвращает дескриптор файла или -1 в случае ошибки. Дескриптор файла используется функциями Read, Write, Seck, Tell, FAbort и Close.

**int Creat(Filename)**

**char \*Filename;**

Создает пустой файл с заданным именем, предварительно стирая любой существующий файл с тем же именем. Новый файл автоматически открывается для чтения/и записи; дескриптор файла возвращается для использования функциями Read, Write, Seek, Tell, FAbort и Close. Возврат значения -1 означает ошибку.

**int Close(FD)**

**int FD;**

Закрывает файл, специфицированный дескриптором файла FD и освобождает FD для использования с другим файлом. Доступ к диску происходит только тогда, когда закрывается файл, открытый для записи; если файл был открыт только для чтения, FD освобождается, но фактически не происходит никакого обращения к CP/M, чтобы закрыть файл. Close не должнa использоваться при буферизованном вводе/выводе файлов. Вместо этого используется FClose. Close возвращает -1 в случае ошибки.

Следует отметить, что все открытые файлы автоматически закрываются по возвращении в пакет времени выполнения из функции Main или, когда вызывается функция Exit. Чтобы предотвратить закрытие открытого файла, используйте функцию FAbort.

**int Read(FD,BUF,NBL)**

**int FD;**

**char \*BUF;**

**unsigned NBL;**

Читает блоки (записи) по 128 байт в количестве NBL в массив BUF из файла с дескриптором FD. Указатель чтения/записи, ассоциируемый с указанным файлом, устанавливается на запись, расположенную вслед за последней прочитанной записью. Каждый вызов Read извлекает данные из того места в файле, на которое установлен указатель при непосредственно предшествующем вызове Read или Write. Функция Seek может использоваться для модификации указателя чтения/записи.

Возвращает число фактически считанных блоков, 0 при обнаружении конца файла, -1 в случае ошибки. Следует заметить, что если вы запросите n блоков данных, когда в файле осталось фактически только x блоков (где 0<х<n), величина х будет возвращена при этом вызове, 0 при следующем вызове (при условии, что вызов Seek не используется) и -1 при последующих вызовах.

**int Write(FD,BUF,NBL)**

**int FD;**

**char \*BUF;**

**unsigned NBL**;

Записывает блоки (записи) по 128 байт в количестве NBL из массива BUF в файл FD. Каждый вызов Write записывает данные на диск, начиная с той точки, на которой остановилось предыдущая операция Read или Write, если только Seek не используется для модификации указателя чтения/записи. Возвращает -1 в случае ошибки или число успешно записанных блоков. Если возвращенное значение неотрицательно, но отличается от NBL, это, вероятно, означает, что вы израсходовали дисковое пространство; это должно рассматриваться как ошибка.

**void Seek (FD,OffSet,Code)**

**int FD;**

**int OffSet;**

**char Code;**

Модифицирует указатель чтения/записи (специфицирующий номер сектора (записи) для последующих операций Read или Write), ассоциированный с файлом с дескриптором FD.

Если Code равно 0, то Seek устанавливает указатель чтения/записи на запись с номером, равным OffSet. Если Code равно 1, то Seek устанавливает указатель чтения/записи на запись с номером, равным текущему значению плюс OffSet (OffSet может быть отрицательным).

Если Code равно 2, то Seek устанавливает указатель чтения/записи на номер записи, равной номеру последней записи в файле плюс OffSet. Естественно, в этом случае значение OffSet должно быть отрицательным, чтобы после этой операции указатель файла указывал на существующую запись. Если Code равен 2 и OffSet - 0, указатель чтения/записи подготовит файл к добавлению записей к нему.

Возврат значения -1 означает, что BDOS был возвращен код ошибки при вызове Seek с Code, равным 2. Функция ErrNo предоставит больше информации о типе произошедшей ошибки. Seek не должна применяться к файлам, открытым для буферизованного ввода/вывода.

**void Tell (FD)**

**int FD;**

Возвращает значение указателя чтения/записи, ассоциируемого с файлом FD. Это число означает, каков номер следующего сектора, который будет записан или прочитан из файла, причем нумерация секторов ведется, начиная с 0.

**int UnLink (Filename)**

**char \*Filename;**

Удаляет заданный файл из файловой системы. Используйте осторожно.

**int Rename(Old, New)**

**char \*Old, \*New;**

Переименует файл обычным образом. Специфицируемый файл не должен быть открыт в то время, когда к нему применяется Rename.

Rename возвращает -1 в качестве сигнала об ошибке.

**int FAbort(FD)**

**int FD;**

Освобождает дескриптор файла FD без закрытия ассоциируемого с ним файла. Если файл был открыт только для чтения, это не окажет никакого воздействия на содержимое файла. Если файл был открыт для записи, то любые изменения, сделанные в текущем экстенте, будут проигнорированы, так как он был открыт последним, но изменения, сделанные в других экстентах, вероятно, останутся. не применяйте вызова функции FAbort к файлу, открытому для записи, если вы не желаете потерять записанные в него данные.

**unsigned CFSize (FD)**

**int FD;**

Подсчитывает фактический размер (в секторах) заданного (открытого) файла без оказания влияния на указатель чтения/записи, ассоциируемый с этим файлом. Следует отметить, что возвращаемый размер будет отражать данные, записанные в новые экстенты, до того как они будут закрыты, в отличие от функции 35 BDOS, используемой обычно для подсчета размера файла.

**int OFlow(FD)**

**int FD;**

Возвращает значение True (не-нулевое), если произошло переполнение в старшем (третьем) байте поля "номер записи при прямом доступе" в FCB, ассоциируемом с данным открытым файлом.

**int ErrNo()**

Возвращает код последней ошибки, обнаруженной при выполнении последней операции по вводу/выводу файлов. Ниже приведен список сообщений об ошибке и соответствующих им кодов.

**char \*ErrMsg(ErrNum)**

**int ErrNum;**

Получив код ошибки, возвращенный ErrNo, эта функция возвращает указатель на цепочку ASC II, описывающую данную ошибку на английском языке. Ниже представлены все возможные коды ошибок и связанные с ними сообщения.

Код Текст

0 не было обнаружено никакой ошибки

1 попытка чтения незаписанных данных

2 выход за пределы области данных на диске

3 не удается закрыть текущий экстент

4 установка указателя на незаписанный экстент

5 не удается создать новый экстент

6 установка указателя за пределы диска

7 задан неверный файловый дескриптор

8 файл не открыт для чтения

9 файл не открыт для записи

10 не осталось ни одного свободного файлового дескриптора

11 файл не найден

12 задан неверный режим в Open

13 не удается создать новый файл

14 установка указателя за запись # 65535

**int SetFCB(FCBAddr,Filename)**

**char FCBAddr[36];**

**char \*Filename;**

Инициирует 36-байтовый блок управления файлом CP/M в области памяти, расположенной по адресу FCBAddr, для файла с именем, содержащимся в строке, заканчивающейся на нуль, на которую указывает Filename. Символы нижнего регистра в строке с именем файла преобразуются в символы Верхнего регистра и добавляется необходимое число пробелов ASC II для заполнения полей имени и расширения FCB. Поля "следующая запись" и "номер экстента" FCB обнуляются. При обнаружении недопустимого символа в имени или расширении Filename знак-"нарушитель" и остаток строки с именем файла будут проигнорированы.

**char \*FCBAddr(FD)**

**int FD;**

Возвращает адрес обычно скрытого от пользователя блока FCB ассоциируемого с открытым файлом с дескриптором FD. Возвращается -1, если FD не является файловым дескриптором открытого файла.

**30. Функции буферизованного ввода/вывода файлов.**

**int FOpen (Filename,IOBUF)**

**char \*Filename;**

**FILE IOBUF;**

Открывает указанный файл для буферизованного ввода и инициирует буфер, на который указывает IOBUF. IOBUF - это указатель на область памяти размером в BUFSIZ байтов, резервируемая для использования функциями буферизованного ввода/вывода. Значение BUFSIZ определяется стандартным файлом заголовка BDS C для ввода/вывода (bdscio.h), который должен быть включен при помощи #include в любую программу, использующую буферизированный ввод/вывод.

Возвращаемое значение - это файловый дескриптор для открытого файла; на самом деле пользователь не нуждается в использовании дескриптора, за исключением проверки отсутствия ошибки, так как значение файлового дескриптора постоянно хранится в буфере ввода/вывода для использования всеми буферизованными функциями ввода/вывода. В случае ошибки возвращается -1.

**int GetC(IOBUF)**

**FILE IOBUF;**

Возвращает очередной байт из входного потока, соответствующего файлу, открытого посредством FOpen с буфером IOBUF. Никакие специальные символы не распознаются: control-z возвращается, как control-z (а не минус 1), cr и lf являются обычными символами и т.д. Вместо аргумента IOBUF для любой функции буферированного ввода могут использоваться значения 0 и 3, чтобы производить ввод из консоли или устройства Reader. GetC (0) эквивалента "GetChar ()". GetC (3) считывает символы с устройства Reader CP/M. При обнаружении ошибки или физического конца файла возвращается -1.При чтении из текстовых файлов посредством GetC как значение 0х1а (CPMEOF), так и -1 (или Error) должны интерпретироваться, как сигнал конца файла, так как некоторые текстовые редакторы CP/M иногда не помещают байт 0х1а (control-z, CPME-OF) в конец текстовых файлов.

**void unGetC(C,IOBUF)**

**char C;**

**FILE IOBUF;**

"Вталкивает" символ с обратно в буфер ввода IOBUF. Следующий вызов GetC для этого файла возвратит этот символ. В определенный момент протолкнуть обратно в буфер можно не более одного символа.

**unsigned GetW(IOBUF)**

**FILE IOBUF;**

Возвращает очередные два байта (т.е., 16-битовое слово) из файла, открытого для ввода с буфером IOBUF, посредством двух последовательных вызовов GetC. В случае ошибки возвращается -1.

**int FCreat(Filename, IOBUF)**

**char \*Filename;**

**FILE IOBUF;**

Создает новый файл с именем, содержащимся в Filename (предварительно удаляя существующий файл с тем же именем) и открывает файл для буферизованного вывода. IOBUF должна указывать на BUFSIZ-байтовый буфер. Возвращает FD для файла или -1 в случае ошибки.

**int PutC (C,IOBUF)**

**char c;**

**FILE IOBUF;**

Записывает байт с в выходной файл с буфером, на который указывает указатель IOBUF. IOBUF должен быть проинициализирован вызовом FCreat. Никаких преобразований не выполняется; текстовые строки могут быть разделены комбинациями cr-lf (для совместимости со стандартным программным обеспечением CP/M) или символами "новая строка" в духе Unix (для более высокой эффективности и простоты). Значения от 1 до 4 можно использовать вместо IOBUF в любых функциях буферизованного вывода, если нужно направить выводимый символ в стандартный выходной поток, стандартное устройство листинга, стандартное перфорирующее устройство, или стандартное устройство вывода. PutC возвращает -1 в случае ошибки.

**int PutW (W,IOBUF)**

**FILE IOBUF;**

**unsigned W;**

Записывает 16 битовое слово W в выходной файл, на буфер которого указывает IOBUF, посредством двух последовательных вызовов PutC. Возвращает -1 при возникновении ошибки.

**int FFlush(IOBUF)**

**FILE IOBUF;**

Сбрасывает на диск содержимое буфера файла - IOBUF, т.e. обеспечивает помещение в файл всех символов, посланных в буфер при последней операции вывода (при условии, что программа не прервана до того, как вызов Exit закроет все файлы). FFlush предназначена для использования файлами буферизованного вывода. Попытка использовать ее для файлов, открытых для ввода не будет иметь никакого эффекта. Следует отметить, что автоматический вызов FFlush происходит всегда, когда выходной буфер заполняется, также как и тогда, когда выходной файл закрывается (через функцию FClose).

**int FClose(IOBUF)**

**FILE IOBUF;**

Закрывает заданный файл буферизованного ввода/вывода (он мог быть открытым для чтения (FOpen) или записи (FCreat). Если файл был открыт для записи, выполняется автоматическая FFlush, чтобы сбросить на диск буфер до закрытия файла. Примечание: до закрытия выходного файла, не забудьте поместить маркер конца текстового файла CP/M (CPMEOF) в конец файла.

**int FPrintF (IOBUF,Format,Arg1,Arg2,...)**

**FILE IOBUF;**

**char \*Format;**

**int Arg1,Arg2,...;**

Совершенно аналогична PrintF за исключением того, что форматированный вывод производится в выходной файл с буфером IOBUF, а не на консоль. Возвращает -1 при возникновении ошибки.

**int FScanF(IOBUF,Format, Arg1, Arg2,...)**

**FILE IOBUF;**

**char \*Format;**

**int Arg1,Arg2,...;**

Действует, как ScanF, за тем исключением, что вводимые символы читаются из входного буфера IOBUF, а не с консоли. Данная версия FScanF требует, чтобы каждая строчка данных сканировалась полностью; любой элемент данных, оставшийся в строке, прочитанной из файла после того, как все спецификации формата были выполнены, будет отброшен. Возвращает число элементов данных, которым успешно присвоено значение или -1, если при чтении файла произошла ошибка.

**char \*FGetS(Str,IOBUF)**

**char \*Str;**

**FILE IOBUF;**

Считывает строку из буфера заданного выходного файла и помещает ее в память по адресу, на который указывает Str. Основана на соглашении CP/M, согласно которому строки завершаются парами символов сr (возврат каретки) и fl (новая строчка) в конце текстовых строк. Чтобы сделать работу с текстовым файлом более простой, FGetS автоматически удаляет cr из каждой пары cr-lf, которая поступает из файла. Любые символы cr, за которыми немедленно не следуют символы lf, остаются нетронутыми. Lf включается в строку, а непосредственно после строки не помещается нулевой байт. Контроль за длиной прочитанной строки не выполняется; Необходимо внимательно следить за тем, чтобы имелось достаточно места в Str, для размещения там максимально длинной строки, ожидаемой при вводе. Нуль возвращается при обнаружении конца файла (eof) независимо от того, встретился ли физический eof (попытка читать после последнего сектора файла) или символ control-z (CPMEOF). В противном случае возвращается указатель на строку (совпадающий с параметром Str).

**int FPutS(Str,IOBUF)**

**char \*Str;**

**FILE IOBUF;**

Записывает строку, заканчивающуюся нулевым байтом, находящуюся в памяти по адресу Str в заданный IOBUF выходной файл. Символы новой строчки преобразуются в комбинации cr-lf, чтобы кухня CP/M содержалась в порядке. Если нуль (нулевой байт) обнаруживается в строке до символа новой строки, то не добавляются завершающие строку на выходе символы cr-lf, что позволяет запись строк по частям.

**31. Особенности реализации.**

BDS C проектировался, как подмножество Си Unix. Поэтому большая часть справочного руководства по языку Си непосредственно относится к BDS C; цель данного приложения - прокомментировать те разделы, которым BDS C не следует в точности.

После предоставления общего списка различий между двумя реализациями автор дает детальное описание, обращаясь к соответствующим номерам разделов книги и описывая то, чем BDS C отличается от того, что написано там.

Вот список самых значительных отклонений BDS C ot Unix Си:

1. Весь исходный файл загружается в основную память сразу, а не по частям. Это лимитирует максимальную длину исходного текста одной функции размерами имеющейся памяти.

2. Компиляция выполняется непосредственно в машинный код 8080, при этом не создается промежуточный файл на языке ассемблера.

3. BDS C написан на языке ассемблера 8080, а не на самом с. будь BDS C написан на самом Си, компилятор был бы в несколько раз больше и скорость выполнения едва ли была бы более высокой. Необходимо помнить, что здесь мы имеем дело с кодом 8080, а не с кодом PDP-II, как в начальной реализации Unix.

4. Типы переменных, такие, как short int, long int, float, double не реализованы.

5. Нет явно объявляемых классов памяти; переменные static и register не существуют, все переменные - или external или automatic, в зависимости от контекста, в котором они объявлены.

6. Сложность объявлений ограничена определенными правилами.

7. Инициализаторы не реализованы.

8. Выделение пространства под строковые переменные производится вручную (нет механизма автоматического размещения и сборки мусора).

Ниже даны комментарии по разделам для справочного руководства по языку Си. В целях краткости некоторые пункты, упомянутые выше, не будут упоминаться в дальнейшем; любые ссылки на типы float, long, static, инициализацию будут игнорироваться.

BDS C создана для микрокомпьютерных систем, базирующихся на процессоре 8080, оснащенных операционной системой CP/M и генерирует бинарный машинный код 8080 (в специальном перемещаемом формате) прямо из исходных текстов программ Си. Естественно, BDS C также может выполняться на любом процессоре, который совместим с 8080, например, Z80 или 8085.

**32. Комментарии.**

Комментарии считаются вложенными по умолчанию; чтобы сделать обработку комментариев BDS C такой, какую делает Unix Си, cc во время компиляции следует задать опцию -c.

**33. Идентификаторы.**

Буквы верхнего и нижнего регистра различаются для имени переменной, структуры, объединения и имен массива, но не для имен функций.

Для компилятора записи

printf("это тест");

и

PriNTf("это тест");

являются эквивалентными.

**34. Ключевые слова.**

Ключевые слова BDS C:

int else

char for

siruct do

union while

unsigned switch

goto case

return default

break sizeof

continue begin

if end

register void

Регистр символов игнорируется для ключевых слов, например while эквивалентна WHilE.

Идентификаторы с именами, совпадающими по написанию с ключевыми словами не разрешаются, хотя ключевые слова включены могут входить составной частью в идентификаторы (например, charflag).

**35. Примечания.**

Оператор "=" используется только для присваивания. Родственный оператор "равняется" обозначается "= =". Будьте внимательны, чтобы их не перепутать; использование не того оператора, который бы вам хотелось употребить, никогда не заставит компилятор выводить какие-либо диагностические сообщения, так как результирующие выражения будут синтаксически правильными, даже если они не дадут желаемого эффекта.

Ключевые слова begin и end могут заменять символы "левая фигурная скобка" и "правая фигурная скобка". Пользователь может использовать компилятор, не имея символов фигурных скобок на своих терминалах.

Эстетически, по крайней мере, такого мнения придерживается автор, лучше использовать фигурные скобки - они создают гораздо более читаемые листинги, чем begin и end и должны быть использованы, где только возможно. Отлов ошибок во время компиляции в некоторых случаях не слишком интеллектуален. Если сс или сс2 извергают набор ошибок, "пасущихся" вокруг одной и той же строки или набора строк, стоит верить только первому сообщению об ошибке в этом "стаде". Есть шанс, что после исправления этой ошибки - остальные уйдут.

Выдача cc2 в сообщениях об ошибках правильных номеров строк не всегда гарантируется. Сс делает некоторую реорганизацию кода. Например, инкрементная часть оператора for физически продвигается вниз за операторную часть. Таким образом, если имеется ошибка в инкрементной части, которую cc не способна обнаружить, сс2 обнаружит ее и сообщит неправильный номер строки. Старайтесь не портить инкрементный раздел операторов for.

Определенные типы ошибок заставляют компилятор прекратить компиляцию и немедленно возвратить управление в операционную систему без анализа остатка исходного текста. Например, это происходит, когда обнаруживается несоответствие открывающих и

закрывающих круглых скобок или пропуск точки с запятой, сбивают компилятор с толку, и он не может самостоятельно исправить эту ошибку. Вместо гадания, в каком же месте произошла ошибка, компилятор прекращает работу.

Следует заметить, что значение ArgC, переданное в функцию Main, всегда положительно и равно числу специфицированных аргументов плюс 1.

Аргументы в командной строке являются строками символов во всех случаях, а не значениями других типов. Чтобы преобразовать к числовому типу, должно быть использовано что-нибудь вроде функции AtoI.

Следующие тонкости нужно помнить, добиваясь оптимальной эффективности:

1. Комментарии динамически удаляются из исходного текста по мере того, как файл считывается с диска; таким образом, нет никаких оправданий (за исключением лени) неадекватному документированию программы.

2. Оператор switch является самым эффективным, когда переменная switch (например xx в "switch(xx)"...) объявляется как char. Целые переменные часто употребляются для сохранения

значения знаков при обработке текстов, при файловом вводе-выводе; присвоение их значения переменным типа char перед большими конструкциями switch поможет сэкономить память и ускорит выполнение.

3. Сase-конструкции оператора switch проверяются в порядке их появления; таким образом, наиболее распространенные значения (или те, которые требуют самого быстрого времени реакции) должны быть расположены вначале.

4. Для достижения максимальной скорости выполнения, задайте сс для компиляции опции -о и -е xxxx, для получении наиболее короткой программы используйте только опцию -е xxxx.

5. Логические выражения в с получают числовое значение 0 (как False) или 1 (как True), если это значение чему-либо присваивается, но это значение теряется при появлении логического выражения в конструкциях условной передачи управления. Это означает, что вы можете воспользоваться преимуществами числовых результатов логических выражений во многих ситуациях. Рассмотрим следующий фрагмент, где нужно приравнять переменную x единице, если а < b или нулю, если а >= b:

if (а<b) x=1; else x=0;

та же самая операция может быть записана, как

x = (a < b);

Это имеет то преимущество, что подвыражение "(a<b)" автоматически выполняет требуемое сравнение и, таким образом, вы избегаете использования двух отдельных операторов присваивания.

6. Появляется возможность краткой записи, в случаях, когда какую-либо переменную необходимо проверить на равенство или неравенство нулю; так как любое выражение можно считать принимающим значение "Тrue", если оно принимает ненулевое значение, часть "=0" выражения "а=0" практически избыточна.

Такие операторы, как

if (a!=0) printf ("a не нуль")

или

if (a= =0) printf ("a есть нуль")

могут быть также записаны, как

if (a) printf ("a не нуль");

и

if (!a) printf ("a есть нуль");

Конечно, такое сокращение не всегда может быть подходящим для подобной ситуации. Если тестируемая переменная используется, как счетчик, и будет принимать множество значений, то выражение "a=0" может быть более понятным для уяснения логики программы. Но в случаях, когда переменная используется, как логический флажок или, когда значение "нуль" рассматривается как некоторое особое значение, тогда более короткая форма является более ясной и может действительно в некоторых случаях привести к более короткому объектному коду.

**36. Типичные ошибки.**

Некоторые из аспектов языка Си вызывают большое количество трудностей, когда с ним начинаешь работать впервые. В этом разделе автор пытается описать эти чувствительные черты Си, на которые пользователи постоянно обращают внимание автора в телефонных звонках и письмах.

Оператор = используется только для присваивания (assignment), в то время, как оператор == используется для проверки истинности соответствующих условий. Два оператора не имеют ничего общего, за исключением литеры, используемой для их представления; они могут затруднить отладку программы при их ошибочном использовании.

Распространенная в Си конструкция - использование операции присваивания в качестве подвыражения другого выражения, в частности, внутри условных операторов. Это может привести к следующим ситуациям:

if((C=GetChar())==Ъ/nЪ)

PrintF("вы напечатали новую строчку")

Здесь начинающий может интерпретировать операцию "=" как проверку условия вместо фактического выражения присваивания.

Теперь рассмотрите следующий фрагмент:

if((C=GetNExt())) {

PrintF("все сделано /n");

break;

}

Условное выражение в этом операторе присваивает значение, возвращенное функцией GetNExt, переменной C, затем проверяет, является ли это значение нулем. Если 0, то печатается "все сделано". Если утомленный программист станет просматривать этот фрагмент, ему может показаться, что Си сравнивается со значением, возвращенным GetNExt ... думаю, вы уловили тонкость в использовании "=" и "==".

**37. Индексация массива.**

Элементы массива длины n в Си имеют элементы, пронумерованные от 0 до n-1. Если вы объявляете массив длиною n и пытаетесь обратиться к элементу Си индексом n, вы в действительности будете обращаться к данным после конца массива. Это часто происходит, когда пользователь мыслит категориями языка Basic, где массивы длины x могут иметь и элемент номер 0 и элемент номер x. Следует заметить, что в Си самая распространенная структура for-цикла аккуратно выполняет n итераций, если цикл записан, как показано ниже:

for (i=0;i<n;i++) {

...

}

И такие циклы являются идеальными для продвижения по массиву. Если же вам нужно иметь массив Си n элементами, пронумерованными от 1 до n, вы должны объявить массив, который имеет на один элемент больше, чем требуется, оставляя 0-й элемент неиспользованным.

**38. Указатели.**

Когда в программе объявляется переменная типа указателя, или внешне или внутри функции, ей не присваивается автоматически никакого значения. Указатель - это просто 16-битовая переменная, которая обычно используется для того, чтобы хранить адрес некоторого другого элемента данных (то есть указывать на него); он должен быть инициирован до использования, как любая другая переменная. Особая ошибка, которую особенно часто встречает автор, - это присваивание значения прямо через неициинированный указатель, т.е. за объявлением

char \*Foо;

следует такой оператор, как

\*Foо='a';

до того, как foо проинициирован. При этом, как правило, происходит нечто непредсказуемое. Оператор присваивания указывает: "поместите символ 'а' в память по адресу, на который указывает переменная Foo". Если указатель Foo не был инициирован, тогда символ 'а' запишется в совершенно произвольную ячейку памяти. Правильным здесь будет объявить буферную область, присвоить ее адрес переменной Foo, затем присваивать значение косвенно через Foo. например, следующая последовательность помещает символ 'а' в буфер [0]:

char Buffer [50], \*Foo;

Foo = &Buffer;

\*Foo = 'a';

**39. Возврат указателей.**

Как только при выполнении функции достигается оператор return, и происходит возврат управления, память, которая была локальной для этой функции (т.е. где хранились все объявленные локальные переменные) освобождается для использования следующей вызванной функцией. Распространенная ошибка - когда некоторая функция (назовем ее Foo) создает кусочек текста в локальной области и возвращает указатель на этот текст. Непосредственно после возврата из Foo текст выглядит неповрежденным, но в дальнейшем (так как пространство, в котором помещается текстовая строка, распределяется для других целей) строка превращается в мусор. Имеется два решения этой проблемы:

а) Foo получает параметр, говорящий, куда поместить содержимое строки текста (в этом случае вызывающая функция должна обеспечить рабочий буфер для Foo);

b) область строки текста объявляется внешней.

каждый метод имеет свои собственные преимущества; передача адреса буферной области разрешает одновременное хранение нескольких символьных строк, полученных на каждый вызов, в различных областях памяти; в то время, как внешняя область для хранения строки сокращает последовательность вызова, требуя передачи на один параметр меньше. Но чтобы вы ни делали, не ожидайте, что данные, которые были локально размещены вызванной функцией, останутся неизменными после того, как выполнится возврат из этой функции.

**40. Формальные параметры.**

Что такое "формальный параметр"? Формальный параметр – это один из аргументов (если таковой есть), поступление которого функция ожидает при каждом ее вызове. Все формальные параметры заданы в начале определения функции в списке, заключенном в скобки, следующем непосредственно вслед за именем функции. Описания всех формальных параметров функции должны быть приведены сразу же после заключенного в скобки списка до первой открытой фигурной скобки, означающей начало тела функции. Любые формальные параметры, не объявленные явно, воспринимаются как значения int (целые). Если формальный параметр случайно объявлен внутри тела функции (внутри фигурных скобок), компилятор будет правильно диагностировать ошибку "повторного объявления". После того как объявления формальных параметров просмотрены, компилятор начинает обработку тела функции; не найдя объявления какого-либо формального параметра, компилятор присваивает ему тип int.

При вызове функции, копии значений всех формальных параметров передаются в функцию. Все такие значения являются 16-битовыми в BDS C. Это означает, что функции, массивы или данные любых других типов, которые имеют размер, отличный от 16 битов, не могут быть переданы прямо в функцию; указатели на эти типы данных могут быть переданы.

Теперь. Что происходит, когда имя массива передается в функцию? Имеется специальный магический механизм для передачи указателей на массивы, что иногда сбивает Си толку, так как из синтаксиса описания интуитивно неочевидно, что был передан действительно указатель.

Например, рассмотрим следующую функцию:

int ArraySum (Array)

int Array [3];

{

return Array[0] + Array[1] + Array[2];

}

Хотя внешне кажется, что ArraySum получает массив из 3 элементов как формальный параметр, в действительности передается только указатель на этот массив. Описание выглядит как если бы передавался весь массив, но если вы измените какой-то элемент в массиве, он изменится также и в вызывающей программе. Существует только одна копия массива.

Другая интересная штучка: вы можете обращаться Си именем массива, как Си простой переменной-указателем внутри вызываемой функции (т.е. присвоить ей адрес другого массива, тогда это имя означает затем базу последнего массива), в то время, как это недопустимо (и приводит к непредсказуемым результатам), когда проделывается над действительным массивом, а не над формальным параметром. книга kernighan & ritchie содержит целую главу о "двойственности" указателей и массивов; в этом механизме заключены самые мощные и самые запутанные аспекты Си.

**41. Вызовы функций.**

Если имя функции используется без списка аргументов, тогда результатом этого подвыражения будет адрес данной функции. Никакого вызова функции не делается, если за именем не следует список параметров, заключенный в скобки, (если список является пустым, он долен быть задан в виде "()"). Например, следующее выражение присваивает адрес конца области внешних данных переменной I.

I = EndExt ();

В то время, как следующее выражение присваивает значение начального адреса функции EndExt переменной I, но только, если EndExt было ранее описано, как имя функции:

I = EndExt;

Следует заметить, что, если EndExt не было ранее описано, когда встречается выражение, подобное второму из вышеприведенных, то компилятор будет правильно распознавать "неописанную переменную" EndExt. В предшествующем примере EndExt неявно описано (в контексте) как имя функции, возвращающей int.

**42. Оверлейная структура.**

Чтобы иметь возможность писать Си-программы, по размеру превышающие имеющуюся физическую память без обращения к Exec или ExecL (которые дают такую возможность, но больше походят на дисциплину подцепления, чем на действительный оверлей), в программу clink был введен набор средств, делающий возможной сегментацию программы. Общая идея: в памяти постоянно располагаются пакет времени выполнения, функция Main и некоторые другие функции, у которых имеется несколько оверлейных сегментов (сегментов наложения). Корневой сегмент контролирует загрузку сегментов наложения в память по старшим адресам и каждый сегмент наложения при загрузке в память где-то выше корневого сегмента может иметь как доступ к точкам входа пакета времени выполнения в корневом сегменте, так и к точкам входа функций любых оверлейных сегментов, расположенных ниже данного, а так же к точкам входа функций корневого сегмента.

Обычно (т.е. когда оверлеи не используются), память при выполнении "си"-прoграммы выглядит так:

base+100h: c.ccc пакет времени выполнения (csiz байтов) адреса (низ)

ram+csiz: начало программного кода... (программный код)...

xxxx-1: конец программного кода

xxxx: область внешних данных (длина y байт) ... (внешние данные) ...

xxxx+y: свободная память, доступная для распределения (Alloc)

????: нижняя точка, до которой добирается стек

стек: локальные данные, параметры функций, результаты промежуточных вычислений

Поясним, что xxxx является первой ячейкой, следующей за программным кодом, а y является количеством памяти, необходимым для внешних переменных.

Чтобы включить оверлеи, мы должны прежде всего решить, где следует размещать загружаемый код. В ранних версиях BDS C блоки локальных данных, размещались в нижней части памяти, непосредственно там, где кончаются внешние данные. Это затрудняло определение первой ячейки памяти, загрузка в которую оверлея не привела бы к катастрофе. Предлагалось для управления оверлеями оставлять достаточно места между концом кода корневого сегмента и началом области внешних данных для размещения загружаемых оверлеев в любом сочетании.

Ныне BDS C размещает всю память для локальных данных в стеке, расположенной в верхней части памяти. И все-таки рекомендуется исходная схема размещения оверлеев; ниже дана модифицированная карта памяти, соответствующая этому методу управления наложениями.

base+100h: c.ccc пакет времени выполнения (csiz байтов) адреса (низ)

ram+csiz: начало кода корневого сегмента ... (код корневого сегмента) ...

zzzz-1: конец кода корневого сегмента

zzzz: начало области оверлеев ... (область оверлеев) ...

xxxx-1: конец области оверлеев

xxxx: область внешних данных (длина y байт) ... (внешние данные) ...

xxxx+y: свободная память, доступная для распределения (Alloc)

????: нижняя точка, до которой добирается стек

стек: локальные данные, параметры функций, результаты промежуточных вычислений

Следует заметить, что точка zzzz располагается там, где загружаются оверлейные сегменты, так, чтобы самый длинный сегмент не достигал xxxx.

В версии 1.5 также можно (но не вполне безопасно) помещать область оверлеев после области внешних данных. Карта памяти для этой альтернативной конфигурации следующая:

base+100h: c.ccc пакет времени выполнения (csiz байтов) адреса (низ)

ram+csiz: начало кода корневого сегмента ... (код корневого сегмента) ...

xxxx-1: конец кода корневого сегмента

xxxx: область внешних данных (длина y байт) ... (внешние данные) ...

xxxx+y-1: конец области внешних данных

xxxx+y: начало области оверлеев (ssss байт) ... (область оверлеев) ...

xxxx+y+ssss-1: конец области оверлеев

xxxx+y+ssss: неиспользованная память

????: нижняя точка, до которой добирается стек

стек: локальные данные, параметры функций, результаты промежуточных вычислений

Следует заметить, что функции распределения памяти (Alloc и SBRK) всегда получают первые участки памяти из области, непосредственно следующей за концом внешних данных. Если вы планируете использовать функции размещения памяти (Alloc, Free, SBRK, RSVSTK) в вашей программе с размещением оверлеев по схеме, приведённой выше, помните, что прежде всего нужно вызвать функцию SBRK с аргументом ssss (размер области наложения). В ином случае "распределитель" начнет выделять память в пределах области наложения.

Чтобы устранить неопределенность, автор предупреждает, что здесь и далее подразумевается использование второй схемы распределения памяти.

А теперь, покончив с общими вопросами, разрешите сказать кое-что о том, как создавать "корневые" и "оверлейные" сегменты для BDS C. Прежде всего нам бы хотелось, чтобы все функции, определенные в корневом сегменте, были бы доступны сегментам наложения. Это достигается тем, что clink записывает файл (с таблицей символов), содержащий все адреса функций, на диск, при компоновке корневого сегмента. Для этого нужно задать опцию "-w". Эта таблица символов будет использоваться позже при компоновке оверлейных сегментов.

При компоновке корневого сегмента используйте опцию -е, чтобы установить начальный адрес области внешних данных. Нужно всегда помнить, что должно быть достаточно места ниже области внешних данных, чтобы разместить самый большой оверлей во время выполнения. Если опция -е опущена, clink устанавливает начало области внешних данных сразу после конца кода корневого сегмента, и вступает в конфликт с областью оверлеев (таким образом, -е может быть опущена при использовании второго варианта оверлеев, как показано на третьей схеме распределения памяти).

Внутри корневого сегмента загружаемый сегмент считывается с диска, для этого нужно задать:

SwapIn(Name,Addr); /\* сегмент считывается ...

но не выполняется \*/

где Addr является ячейкой, следующей за последним байтом кода корневого сегмента. Вы можете найти это значение, откомпоновав один раз корневой сегмент без задания опции -е и прочитав статистику опции -s, выданную на консоль после компоновки. Чтобы действительно выполнить сегмент, вы должны вызвать его, используя переменную типа "указатель на функцию". Ниже представлен пример. мы объявим переменную "указатель на функцию" с именем PtrFn, загрузим в память сегмент по имени ovll в ячейку 3000h, и вызовем этот сегмент. Последовательность будет следующей:

int (\*PtrFn) (); /\* может быть любого типа \*/

ptrfn=0x3000;

if (SwapIn ("ovll",0x3000)==Error) /\* проверка отсутствия

ошибки при загрузке \*/

(\*ptrfn)(args...); /\* если нет, вызов сегмента \*/

Следует заметить, что оверлей может не возвращать значений после того, как его вызвали, но указатель на функцию должен быть описан с каким-либо типом. Используйте int, если больше ничего не приходит в голову. Когда сегмент вызывается, как выше, управление передается в функцию Main оверлея. Нет никакой причины ограничивать набор аргументов функции Main оверлея парой "ArgC" и "ArgV", т.к. нет ничего особенного в функции Main, кроме того, что она вызывается первой. Функция Main в оверлее - это единственная точка входа, допустимая в оверлейном сегменте.

Теперь мы знаем все, кроме порядка создания собственно оверлейных сегментов. Оверлей - это, в основном, нормальная "Си"-программа, у которой, также, как и у корневого сегмента, есть функция Main, за тем исключением, что пакет времени выполнения c.ccc не подключается в начале оверлейного сегмента (пакет времени выполнения c.ccc в корневом сегменте может быть в коллективном пользовании). другое различие между оверлейным и

корневым сегментом - это адрес загрузки; в то время, как корневой сегмент всегда загружается на основание TPA, сегмент наложения может загружаться, где угодно. После того, как вы

оттранслировали оверлейный сегмент, вы даете специальную форму команды clink, чтобы компоновать его:

A>clink <сегм> -v -l xxxx -y <симв> [-s ...] <вк>

где <сегм> - это имя crl-файла, содержащего оверлей, -v означает для clink, что должен быть создан сегмент наложения (поэтому c.ccc не присоединяется), а -l xxxx (за буквой "эль"

следует шестнадцатеричный адрес) означает адрес загрузки для сегмента. Опция -y заставляет использовать для вызовов функций адреса, определенные в файле <симв> в таблице символов, созданной корневым сегментом. Если опция опущена, clink использует "свежие" копии функций, таких как "PrintF" и "FOpen" и т.д., даже если они были включены в корневой сегмент. При чтении символьной таблицы корневого сегмента гарантируется, что все программы, уже подкомпонованные, будут доступны оверлею. Корневой сегмент, однако, не может знать о функциях, принадлежащих оверлею, через использование символьной таблицы. Это потребовало бы использования компоновки с перекрестным обращением, но эти возможности лежат за пределами данного пакета.

При компоновке оверлея (сегмента наложения) вы можете также специфицировать -s, чтобы выдать на консоль статистику и -w, чтобы записать расширенную таблицу символов, содержащую не только символы, считанные из sym-файла корневого сегмента, но также и собственные символы загрузочного сегмента.

Пора рассмотреть маленький пример: допустим, у вас есть программа root.c, которая будет загружать и выполнять seg1.c, а затем поверх seg1.c загрузит seg2.c. root.com загружается по адресу 100h и заканчивается, скажем, по адресу 3000h. Мы загрузим сегменты с ячейки 3000h и установим начало области внешних данных на 5000h (при этом будем предполагать, что сегментов длиннее 2000h нет).

Вот как будет выглядеть компоновка:

A>clink root -e 5000 -w -s <вк>

это говорит clink, что root.com должна быть корневым сегментом (так как нет опции -v), внешние данные начинаются с адреса 5000h, необходимо создать файл с таблицей символов под именем root.sym, а на консоль должна быть выведена статистика (-s).

Компоновка каждого оверлея выполняется следующим образом:

A>clink seg1 -v -l 3000 -y root -s -o seg1. <вк>

Это говорит clink, что seg1.com должна быть оверлейным сегментом (-v), адрес загрузки которого - 3000h, фaйл таблицы символов под именем root.sym должен быть сканирован для адресов определенных ранее функций, после компоновки должна быть выдана

статистика, а итоговый файл должен быть записан под именем seg1.com, чтобы избежать его случайного вызова как команды CP/M).

**43. Препроцессор ассемблера.**

Casm - это препроцессор, который получает на входе исходный файл на языке Си-ассемблера с расширением ".csm" (сокращение от с assembly) в специальном формате и записывает файл ".asm", который может затем быть ассемблирован стандартным, вездесущим ассемблером CP/M (аsм.сом). Casm выясняет, какие команды языка ассемблера используют перемещаемые адреса и вставляет соответствующие псевдо-операции и дополнительные коды операций в получаемый "asm"-файл, поэтому последующее ассемблирование образует непосредственно формат crl. Кроме того, выполняются элементарные логические проверки: обнаруживаются и сообщаются неопределенные и/или повторно определенные символы; дублирование имен в разных функциях допускается, т.к. они преобразуются в уникальные имена, поэтому asm не обнаружит ошибку.

**44. Опции командной строки.**

"-с" - выключает подавление комментариев на входе и на выходе. По умолчанию casm удаляет все комментарии из входного файла при чтении его и не помещает никаких комментариев в образуемый ассемблерный текст формируемого asm-файла. Если задана опция -с, исходные комментарии сохраняются и casm добавляет свои собственные комментарии в получаемый файл.

"-f" - помечает флажками старые макро-команды из библиотеки cmac.lib, чтобы помочь пользователю преобразовать старые исходные файлы на языке ассемблера в формат csm.

"-o имя" - присваивает выходному файлу имя имя.asm. Обычно выходной файл получает имя, состоящее из имени входного csm-файла и расширения ".asm". Псевдооперации, которые воспринимаются casm, как специальные управляющие команды в пределах файла "csm", следующие:

function <имя>

Каждая функция должна начинаться псевдо-операцией function <имя>, где <имя> является именем, которое будет использоваться для этой функции в оглавлении crl-файла. В этой строке не допускается никакой другой информации. Внимание: при обработке casm правильно не обрабатывает имена функций, содержащих символ "подчеркивающая черта", хотя компилятор, линковщик и библиотекарь Си обращается с такими именами вполне корректно. Примечание: в процессе обработки весь исходный текст переводится вверхний регистр.

external <список>

Если функция вызывает другую, закодированную на Си или на ассемблере, сразу за командой function должна следовать псевдооперация external, в которой перечислены эти функции, причем в списке можно употребить несколько имен, а в одной функции может быть несколько строк external. В строках external допустимы только имена функций; имена данных (такие, как переменные "external", определённые в программах Си) не могут быть заданы в операторах "еxternal".

endfunc, endfunction

эта операция (обе формы допускаются) должна быть помещена после тела данной функции. Имя функции не должно указываться в качестве операнда. Три вышеперечисленные псевдооперации должны обязательно находиться среди команд языка ассемблера в ".csm"-файле.

include <имя файла>

Эта операция вызывает вставку заданного файла в текущую строку выходного файла. Обычно в начале csm-файлов помещают инструкцию #include "bds.lib", поэтому имена стандартных подпрограмм пакета времени выполнения распознаются casm и не интерпретируются как неопределенные локальные ссылки. Вид исходного текста для casm:

include "bds.lib"

function <имя-1>

[ external <имя1>, [ ... ] ]

код для функции 1

еndfunc

function <имя-2>

[ external <имя1>, [ ... ] ]

код для функции 2

endfunc

end

Дополнительные замечания:

1. Если ассемблерная команда используется с меткой, имя метки должно начинаться в 1-й колонке. Если имя метки не начинается в колонке 1, casm не воспримет его, как метку и преобразование не будет произведено правильно.

2. Ссылки вперед на equ-символы в выполняемых командах не разрешаются, однако допустимы ссылки вперед на перемещающиеся символы. Это происходит из-за того, что casm - однофазный процессор и как только в команде встречается ранее не известный символ, casm принимает, что символ является перемещающимся, и генерирует параметр перемещения для команды.

3. include действуют только для одного уровня вложенности.

4. Когда необходимо специфицировать перемещаемое значение в команде dw, оно должно быть единственным значением, заданным в этом конкретном операторе dw, в противном случае преобразование будет произведено неправильно. Другими словами, только один 16-битовый перемещаемый элемент разрешается для каждого оператора dw.

5. Символы, используемые в именах, могут быть только буквенно-цифровыми; знак денежной единицы также разрешается, но его использование может привести к конфликту с метками, генерированными casm.

6. Файл "hex", созданный asm после ассемблирования выхода casm, не может быть преобразован в бинарный файл, путем использования команды CP/M load, вместо нее должны использоваться ddt или sid для чтения файла в память, и затем должна быть введена команда save для записи на диск "crl"-файла. Casm вставляет строчку в конец "asm"-файла, заканчивающуюся символами ".", поэтому эта строка будет отмечена флажком как ошибочная. Тогда пользователь может посмотреть на значение, напечатанное слева от нее на полях, чтобы увидеть точно, сколько 256-байтовых блоков необходимо задать для save после использования ddt или sid, чтобы записать файл на диск. Причина, по которой load не может быть использована, заключается в том, что casm выдает код для генерации указателя файла crl в конце файла asm, используя псевдо-оператор "org", чтобы установить счетчик команд обратно на начало TPA. Команда load прерывает работу с выдачей сообщения "inverted load address", когда встречаются подобные ситуации. Вместо того, чтобы casm записывала оглавление в отдельный файл, а затем добавляла файл с командами в конец этого файла с оглавлением, автор решил потребовать от пользователя использовать команду save.

**45. Файловый ввод.**

Библиотечные функции, имеющиеся в BDS C для выполнения ввода/вывода, делятся на две основные категории: первичные, или функции ввода/вывода низкого уровня, и буферизованные функции ввода/вывода.

Первичные функции, для достижения лучших характеристик выполнения, закодированные на языке ассемблера, по существу являются расширенным интерфейсом BDOS, которая и выполняет все операции ввода/вывода. Объем данных, передаваемых во время выполнения первичных функций ввода/вывода, всегда кратен размеру одного логического сектора CP/M (128 байт).

Буферизованные функции (написанные на Си) обеспечивают байт-ориентированную последовательную файловую систему ввода/вывода, особенно удобную при организации фильтров. Они позволяют читать и записывать данные порциями наиболее удобных размеров, благодаря тому, что скрытые механизмы управляют всеми операциями буферизации и обмена с дисками. Таким образом, буферизованные функции ввода/вывода являются более удобными в работе, чем первичные функции, но их использование более накладно в смысле скорости выполнения и расходования памяти за счет обслуживания буферных областей.

Так как первичные функции ввода/вывода - это строительные блоки, из которых строятся буферизованные функции, автор прежде всего опишет детально первичные функции, а затем перейдет к буферизованным функциям.

**46. Первичные функции файлового ввода/вывода.**

Для всех первичных функций ввода/вывода характерно использование для идентификации обрабатываемых файлов файловых дескрипторов. Файловый дескриптор, или FD, - это небольшое целое число, которое приписывается файлу, когда этот файл открывается или создается, и остается ассоциированным с файлом до тех пор, как он будет закрыт. FD создается при помощи вызова функций Open и Creat.

Использование этих функций:

FD = Open(Filename,mode);

FD = Creat(Filename);

Open используется для открытия уже существующего файла (обычно непустого) для чтения, записи или того и другого одновременно. Creat используется для создания нового файла и открытия его для чтения и записи. В обоих случаях функция возвращает FD, если открытие или создание файла прошло успешно. В случае ошибки (специфицированный файл не может быть открыт или создан), возвращается значение Error (-1), после чего может быть вызвана функция ErrNo, чтобы выяснить, почему файл не может быть открыт.

Все другие первичные функции требуют указания FD, чтобы специфицировать обрабатываемый файл (за исключением UnLink и Rename, параметром которых является имя файла). Две важные первичные функции ввода/вывода Read и Write выполняют обмен данными между диском и памятью 128-байтовыми логическими секторами. Способ их обычного использования:

I = Read(FD,Buffer,Sects);

J = Write(FD2,Buffer2,Sect2);

Первый оператор пытается считать Sects секторов данных из файла, FD которого указан, в память по адресу Buffer. Второй оператор пытается записать Sects2 секторов данных из памяти, начиная с ячейки Buffer2, в дисковый файл, дескриптором которого является FD2. Если не происходит ошибки (что случается тогда, когда задано неверное значение FD или делается попытка читать после конца файла), Read и Write должны вызвать немедленную активность дисковода. Одно из основных различий между первичным и буферизованным вводом/выводом: первичные функции вызывают немедленную активность дисковода, если, конечно они могут быть выполнены, в то время, как буферизованные функции обращаются к диску только тогда, когда буфер заполняется (во время записи) или опустошается (во время чтения).

С каждым файлом, открытым для первичного ввода/вывода, ассоциируется специальный указатель, он называется указателем чтения/записи ("r/w указатель") и скрыт от пользователя. Этот указатель содержит номер очередного сектора, который должен быть записан в файл или считан из файла. Сразу же после открытия файла указатель r/w устанавливается в 0 (первый сектор файла). Он автоматически увеличивается после каждого из вызовов Read и Write, при этом приращение равно количеству успешно переданных секторов. Таким образом, каждая операция по передаче данных "знает", в каком месте остановилась предыдущая операция. Значение указателя r/w файла возвращается функцией Tell и может быть модифицировано при помощи функции Seek.

Для иллюстрации использования первичного ввода/вывода в программе, построим утилиту, копирующую файл.

Формат команды для этой утилиты, (которую мы назовем "copy") будет таким:

A>copy файл1 файл2 <вк>

"Сopy" возьмет файл с именем "файл1" и создаст копию, с именем "файл2". Так как мы хотим создать первоклассную утилиту, необходимо обеспечить диагностику ошибок (таких, как переполнение диска, невозможность найти исходный файл, и т.д.). Сюда войдет и проверка правильности числа параметров, введенных в командной строке. Дадим вначале краткое описание программы в форме псевдокода.

Copy (File1,File2)

(

eсли (не было задано ровно 2 аргумента)

сообщение об ошибке и выход

если (не удалось открыть файл 1)

сообщение об ошибке и выход

если (не удалось создать файл 2)

сообщение об ошибке и выход

пока (не закончен файл 1) (

прочитать большой кусок из

файла 1 и записать его в файл 2;

если (произошла какая-то ошибка)

сообщение об ошибке и выход

)

закрыть файлы;

)

Ниже представлена программа на Си для выполнения операции копирования.

#include "bdscio.h" /\* стандартный файл заголовка \*/

#include "types.h" /\* новые типы данных \*/

#define BufSects 64 /\* буферизация до 64 секторов в памяти \*/

fd FD1,FD2; /\* файловые дескрипторы для двух файлов \*/

byte Buffer[BufSects \* SecSiz]; /\* буфер передачи \*/

Main(ArgC,ArgV)

int ArgC; /\* счетчик аргументов \*/

char \*ArgV[]; /\* вектор аргументов \*/

{

int OkSects; /\* временная переменная \*/

/\* убедимся, что задано ровно 2

аргумента \*/

if (ArgC!=3)

PrErr("Вызов: A>copy Filel File2 <вк>\n");

/\* попытаемся открыть файл; прервемся в

случае ошибки \*/

if ((FD1 = Open(ArgV[1],0)) == Error)

PrErr("невозможно открыть файл %s\n",ArgV[1]);

/\* попытаемся создать второй файл;

прервемся в случае ошибки \*/

if ((FD2 = Creat(ArgV[2])) == Error)

PrErr("невозможно создать файл %s\n",ArgV[2]);

/\* сейчас мы готовы пересылать \*/

while (OkSects = Read(FD1,Buffer,BufSects)) {

if (OkSects == Error)

PrErr("ошибка чтения\n");

if (Write(FD2,Buffer,OkSects) != OkSects)

PrErr("ошибка записи\n");

}

/\* копирование завершено. теперь закроем файлы \*/

Close(FD1);

if (Close(FD2) == Error)

PrErr("не удалось закрыть файл %s\n",ArgV[2]);

PrintF("копирование завершено\n");

{

PrErr(Format,Arg) /\* вывод сообщения об ошибке и

{ прекращение работы \*/

PrintF(Format,Arg);

FAbort(FD2); /\* освободить дескриптор файла и

ликвидировать результаты файло-

вых операций \*/

Exit(); /\* возврат в CP/M \*/

}

Теперь давайте рассмотрим программу подробнее. Сначала идет объявление: нам необходим файловый дескриптор для каждого файла, используемого в процессе копирования и большой массив для буферизации данных, так как пересылаемые части файлов размещаются в памяти. Размер буфера вычисляется, как размер сектора (SecSiz, определенный в вdscio.h), умноженный на число буферизуемых секторов (константа BufSects, определенная в начале программы).

В функции Main мы прежде всего убеждаемся, что в командной строке введено правильное число параметров. Так как параметр "ArgC", передаваемый пакетом времени выполнения в Main, всегда равен числу заданных параметров плюс 1, мы проверяем, равен ли он трем (т.е. было ли введено два параметра). Если ArgC не равен трем, мы вызываем PrErr, чтобы вывести сообщение об ошибке и прервать программу. PrErr передает свои аргументы функции PrintF, прекращает операции на выходном файле и возвращает управление на командный уровень.

Если проверка ArgC произведена успешно, настало время попробовать открыть файлы. Следующий оператор открывает файл для чтения, заносит файловый дескриптор, возвращенный Open, в переменную "FD1" и вызывает остановку программы, если произошла ошибка при выполнении Open. Это можно сделать "одним махом" благодаря возможностям вычислителя выражения Си;

Прежде всего выполняется вызов Open, затем возвращенное Open значение присваивается переменной "FD1" и производится проверка, не равно ли это значение константе Error. Если значение не равняется Error, значит файл был открыт правильно и управление перейдет к следующему оператору; в противном случае выполняется вызов PrErr. Создание выходного файла выполняется подобным образом и опять с вызовом PrErr, если попытка создания файла возвращает значение Error.

Проделав все подготовительные мероприятия, можно начать копирование данных (наконец-то). Каждый раз в начале цикла while мы читаем столько данных в память из исходного файла, сколько можем (до BufSects секторов). Функция Read возвращает число успешно считанных секторов; оно лежит в диапазоне от 0 (ситуация "конец файла") до числа запрашиваемых секторов (в данном случае BufSects) и, кроме того, в случае стихийного бедствия (открылась дверца дисковода и т.п.), принимает значение Error. Независимо от того, какое значение возвращено, oно присваивается переменной "OkSects" для дальнейшего использования. В случае, когда оно равняется нулю, что означает EOF, происходит выход из цикла while. В противном случае, мы входим в цикл и пытаемся записать данные, которые только что были считаны. Прежде всего мы хотим убедиться, что не произошло никакой грубой ошибки; для этого производится проверка того, было ли программой Read возвращено значение Error. Если ошибки нет, вызываем Write, чтобы сбросить данные в выходной файл. Если не удастся записать точно, то число секторов, которые мы хотели записать, то выдаем сообщение об ошибке (большинство ошибок записи вызываются переполнением диска). Если Write завершается успешно, мы возвращаемся в начало цикла и пытаемся прочитать еще некоторое количество данных. Этот процесс продолжается до тех пор, пока не будут считаны и записаны все данные, в этом случае функция Read возвращает нуль и выполняется выход из цикла while.

Последнее, что необходимо сделать - это закрыть файлы; при этом мы должны убедиться, что файл был закрыт правильно.

**47. Функции буферизованного ввода/вывода файлов.**

Первичные функции ввода/вывода, представленные в предыдущем пункте, наиболее полезны, когда необходимо манипулировать большими объемами данных, скажем, в случае создания копии файла. Это усложняет работу с данными произвольного размера. Рассмотрим, например, строчку текста: строчка, состоящая из литер ASC II, может изменяться по размеру от 1-го байта (в случае пустой строки, представленной только завершающим нулем) до 130 байтов или даже больше. Удобный метод чтения и записи этих текстовых строчек в файлы и из файлов будет очень полезным для обработки текста. В идеальном случае нам бы хотелось вызвать некую функцию, посылая ей какой-то файловый дескриптор вместе с указателем на строку текста и иметь функцию записи строчки текста в файл, вслед за последней записанной строчкой. Также, чтобы предотвратить расход времени на доступ к диску каждый раз, когда записывается строчка, было бы хорошо, если бы наша функция буферизировала ряд строчек и записывала их все на диск сразу по заполнении буфера. Аналогично, должна быть функция для чтения текстовой строки из файла в память; здесь в большой степени увеличится производительность, если невидимый буфер будет управляться функцией чтения текстовой строчки, вследствие чего активность дисковода сократится до минимума. Функции FPuts и FGetS действительно имеются в стандартной библиотеке.

Основное понятие буферизованного ввода/вывода - это структура, называемая "буфер ввода/вывода". Элементы этой структуры - большой массив char, чтобы хранить передаваемые данные, и несколько указателей и дескрипторов, чтобы знать, "что происходит" в массиве данных буфера, а именно: файловый дескриптор, чтобы идентифицировать файл для первичных операций ввода/вывода, указатель позиции в массиве данных, чтобы знать, где должен быть записан следующий байт, счетчик, который указывает, сколько байтов данных или свободного пространства (в зависимости от того, записываете вы или считываете) осталось до того, как возникнет необходимость очередного чтения или записи в буфер, и, в заключение, вспомогательный набор битов, который содержит разнообразную информацию, например, используется ли буфер для ввода или для вывода. Функции буферизованного ввода/вывода используют указатели на буферы ввода/вывода, как идентификацию обрабатываемого файла, в то время, как первичные функции ввода/вывода файлов используют файловые дескрипторы.

Имеется шесть функций, выполняющих операции по вводу/выводу в файл одного байта данных (т.е. char). Другие буферизованные функции ввода/вывода (такие, как FPuts и FGetS ) делают свою работу, используя шесть этих "основных" функций.

Для чтения файлов имеются функции FOpen, GetC, и FClose. FOpen вызывается, чтобы открыть уже существующий дисковый файл, ассоциирует его с буфером ввода/вывода, предоставленного пользователем, и инициирует этот буфер для получения данных из файла. GetC извлекает один байт из буфера, заботится при этом о пополнении буфера, если он пуст, и возвращает специальное значение EOF (-1), кoгда достигнут физический конец файла. FClose закрывает файл, ассоциируемый с буфером ввода/вывода, и освобождает буфер для использования с другим файлом.

Для записи файлов предназначены функции FCreat, Puts, FFlush и FClose. FCreat создает новый файл и готовит соответствующую структуру буфера ввода/вывода для получения выходного файла. Данные записываются в буфер посредством вызовов PutC, по одному байту; когда вызов PutC вызывает заполнение буфера, буфер сбрасывается на диск. Когда все данные записаны в файл, FClose "закрепляет достигнутое", закрывая ассоциируемый файл. Для файлов, открытых для записи, FClose прежде всего вызывает FFlush, чтобы сбросить ("flush") содержимое незаполненного буфера ввода/вывода в дисковый файл, до того, как его закрыть.

Единственными функциями, которые сами записывают и считывают данные, являются GetC и PutC; такие функции, как FGetS, FPuts, FPrintF и т.д. выполняют запись и чтение при помощи GetC и PutC.

Внимательное рассмотрение заголовка файла bdscio.h показывает, что число секторов, используемых для буферизации, по умолчанию равно 8 и что это значение может быть изменено пользователем для оптимальной работы на различных системах. Например, если вы используете BDS C на системе CP/M с форматом диска, сектор которого имеет размер 1024-байт, тогда в буферизации восьми 1024-байтовых секторов, выполняемой функциями буферизованного ввода/вывода, нет необходимости, и применение буферизации 1 сектора сэкономит много памяти, не вызывая заметных потерь в скорости выполнения. А вот на системах CP/M, работающих с 8" дискетами со 128-байтовыми секторами, принятая по умолчанию буферизация по 1кбайт действительно ускоряет дело.

Давайте рассмотрим пример. Следующая программа выводит текстовый файл на консоль, генерируя номера строк на полях с левой стороны:

/\*

pnum.c: программа для вывода на консоль текстового файла

с автоматической генерацией номеров строк. \*/

#include "bdscio.h"

#include "types.h"

Main(ArgC,ArgV)

char \*ArgV[];

{

file IBuf; /\* буфер ввода/вывода \*/

char LinBuf [MaxLine]; /\* временный буфер для строки \*/

int LineNo; /\* номер строчки \*/

if (ArgC!=2) { /\* убедимся, что файл задан \*/

PrintF("ввод команды :A>pnum filename <вк>\n");

Exit ();

}

if (FOpen (ArgV[1],IBuf) == Error ) {

PrintF("нельзя открыть %s\n", ArgV {1};

Exit ();

}

LineNo=1; /\* инициация номера строки \*/

while (FGetS(LinBuf,IBuf))

PrintF ("%3d: %s",LineNo++,LinBuf);

fclose(IBuf);

}

Объявление IBuf обеспечивает область буфера ввода/вывода для использования с FOpen, GetC, FClose.

После проверки числа аргументов и открытия специфицированного файла для ввода, вся реальная работа происходит в одном простом операторе while. Функция FGetS читает строку текста из файла и помещает ее в массив LinBuf. До тех пор, пока не встретится конец файла, FGetS будет возвращать ненулевое значение (True), и тело оператора while будет обрабатываться. тело состоит из одного вызова PrintF, в котором печатается текущий номер строчки, за которым следует двоеточие, пробел и текущая строка текста. После использования значения LineNo, оно увеличивается на 1 (оператором ++) для следущей итерации. Цикл считывания и печати выполняется до тех пор, пока FGetS возвращает нуль; в этой точке цикл while прекращается и FClose завершает работу программы.

В качестве последнего примера - программа из разряда программ-фильтров. Обычно программа-фильтр считывает входной файл, выполняет всякого рода преобразования в нем и записывает результат в новый выходной файл. Преобразования могут быть довольно сложными (компилятор Си) или тривиальными, например, перевод входного файла в верхний регистр. Так как стоимость печати сейчас довольно высока, давайте на некоторое время отложим рассмотрение Си-компилятора и разберем программу перевода в верхний регистр.

/\*

upper.c: программа для преобразования произвольного вход-

ного текстового файла в верхний регистр \*/

#include "bdscio.h"

#include "types.h"

Main (ArgC,ArgV)

char \*ArgV[];

{

file IBuf,OBuf;

int c;

if (ArgC=3) {

PrintF("Вызов: A>upper <источник> <приемник> <вк>\n");

Exit ();

}

if (FOpen (ArgV[1], IBuf) == Error) {

PrintF ("не могу открыть %s\n", ArgV [1]);

Exit ();

}

if (FCreat (ArgV [2], OBuf ==Error) {

PrintF ("не могу создать %s\n", ArgV [2]);

Exit ();

}

while ((c = GetC (IBuf))=EOF&&c=CPMEOF) {

if (PutC (ToUper(c), OBuf) == Error)

PrintF ("ошибка записи; диск, вероятно, заполнен\n");

Exit ();

}

PutC (CPMEOF, OBuf);

FClose (OBuf);

FClose (IBuf);

}

На этот раз у нас два буферизованных файла для ввода/вывода: выходной файл и входной файл. Первая задача - проверить, было ли введено правильное число параметров в командной строке. мы ожидаем два параметра: имя существующего входного файла и имя выходного файла, который необходимо создать. Затем вызываются FOpen и FCreat, чтобы открыть и создать два файла для буферизованного ввода/вывода. Если это удается, входим в основной цикл, и работа начинается.

При каждой итерации цикла один байт извлекается из входного файла и сравнивается с двумя возможными обозначениями конца текстового файла: EOF и CPMEOF. Обычно последним в текстовом файле должен быть символ CPMEOF (control-z). Но некоторые текстовые редакторы не помещают знак CPMEOF в конце файла, если файл заканчивается точно на границе сектора; в этом случае CPMEOF никогда не будет извлечен, и для выявления конца файла нужно выполнить сравнение со значением физического конца файла (EOF). Трудности, вызванные этим довольно запутанные.

Значение EOF, возвращенное GetC, является -1, оно должно быть представлено как 16-битовое значение, т.к. переменные char в BDS C не могут иметь отрицательные значения. Поэтому переменная "с" объявляется как int вместо char в выше описанной программе; если она была объявлена как char, тогда результат подвыражения

c = GetC (IOBuf);

будет иметь тип char, и оно никогда не будет равно EOF. Если GetC возвратит EOF, "c" получит значение 255 (интерпретация char 8 младших битов значения EOF). Поэтому "с" объявляется, как int, и сравнение с EOF имеет смысл. Это странно, так как "c" используется в программе для хранения байта и было бы хорошо объявить ее, как переменную типа char. На самом деле есть способ добиться универсальности: если вместо EOF в условном выражении употребить 255, "с" следует объявить, как char и программа будет работать. За исключением того случая, когда во входном файле встретится байт 0ffh (десятичное 255). Если есть уверенность, что в файле нет 0ffh-байтов, это допустимо, но могут быть исключения. Нет закона, где бы говорилось, что команды фильтрации могут быть применены только для текстовых файлов. Захотим ли мы прекращать обработку, когда в файле встретится байт ff? Нет, исходный метод является самым общим.

Определив, что конец файла не встретился, в работу включается тело оператора while. Здесь мы используем ToUpper, чтобы преобразовать символ, полученный из GetC, в верхний регистр, и PutC, чтобы записать полученный байт в выходной файл. Выполнение прекращается, если PutC возвращает Error.

Как только обнаружено условие "конец файла", мы записываем в конец выходного файла символ CPMEOF (control-z). Наш фильтр написан так, что CPMEOF будет добавлена к выходному файлу независимо от того, завершился входной файл символом CPMEOF, или

нет. Наконец, FClose используется, чтобы закрыть входные и выходные файлы.

**48. Консольный ввод/вывод.**

В этом разделе автор пытается дать разъяснения по механизму консольного ввода/вывода CP/M и показать, как лучше использовать преимущества этого механизма в программах на BDS C.

Основное внимание будет уделено использованию библиотечных функций BIOS и BDOS для выполнения ввода/вывода непосредственно через BIOS и BDOS.

Одной из причин прямого обращения к BIOS CP/M для консольного ввода/вывода вместо использования функций GetChar/PutChar, имеющихся в стандартной библиотеке является желание преодолеть неожиданные трудности обработки определенных символов ASC II в BDOS CP/M и функциях GetChar/PutChar (которые используют вызовы BDOS для своих целей). Их удобно применять в телекоммуникационных программах, играх или других программах, где требуется более "прозрачная" работа с консолью, чем та, которую обеспечивают функции GetChar и PutChar.

**49. Элементарный консольный интерфейс.**

Давайте взглянем на то, что происходит в действительности во время консольного ввода/вывода и на то, как с ним работать.

Самый низкий (самый простой) уровень программного обеспечения, управляющего консолью - это раздел BIOS (основная система ввода/вывода) CP/M. Имеются следующие подпрограммы BIOS, записывающие и считывающие символы с консоли или на консоль: ConSt (проверить консольный статус), ConIn (ждать нажатия клавиши на клавиатуре, затем ввести символа) и ConOut (вывод символа на консоль). Библиотека BDS C содержит функцию BIOS, делающую очень простым доступ из программ на Си к подпрограммам BIOS.

Векторы 2,3 и 4 BIOS используются, чтобы получить прямой доступ к консоли. Выражение BIOS (2) обеспечивает вызов подпрограммы BIOS ConSt, которая возвращает ненулевое значение, когда имеется символ на вводе консоли, в противном случае возвращаемое значение - нулевое. Чтобы произвести фактическое чтение символа после того, как BIOS (2) возвестил о готовности консоли подождать до тех пор, пока символ будет готов, и затем его прочитать, используйте BIOS (3), вызывающую подпрограмму ConIn, возвращенное при этом значение и есть символ, введенный с консоли. Чтобы выдать символ c на консоль, обратитесь BIOS (4,c), вызывающую ConOut. Следует заметить, что подпрограммы BIOS не в курсе о том, что c-программы представляют комбинацию возврат каретки-перевод строки одним символом "новая строка" ('\n'). Вызов BIOS (4,'\n') вызовет печать только одного этого символа (ASC II десятичное значение 10) на консоль без возврата каретки. При использовании прямого консольного ввода/вывода вы должны послать и возврат каретки ('\r') и перевод строки (новую строку) ('\n') в подпрограмму ConOut, чтобы перейти к началу новой строки на консоли. Делается это так:

BIOS (4,'\r'); /\* возврат каретки в ConOut \*/

BIOS (4,'\n'); /\* перевод строки в ConOut \*/

Только использование функции BIOS для ввода/вывода на консоль дает уверенность, что программы на Си, использующие полноеуправление консолью, будут мобильны

**50. BDOS все усложняет.**

Следующий (более высокий) уровень интерфейса (над BIOS), выполняющий ввод/вывод на консоль - это BDOS (основная дисковая операционная система). Поскольку имеются три основные подпрограммы BIOS для интерфейса с консолью, существуют три подобные, но "более высокого уровня" функции BDOS для выполнения подобных задач. эти функции BDOS, каждая из которых имеет свой собственный кодовый номер, в отличие от BIOS, следующие: "консольный ввод" - получение символа с консоли (функция 1 вdos), "консольный вывод" - вывод символа на консоль (функция 2 BDOS), "получить консольный статус" - определить, имеется ли символ на вводе консоли (функция 11 BDOS).

Стандартные библиотечные функции GetChar и PutChar выполняют свои задачи при помощи BDOS. Которые в свою очередь выполняют свои операции при помощи вызовов BIOS, что приводит к некоторым неприятным ситуациям. Операции BDOS делают все для вас, даже то, о чем вы и не знаете. Например, если BDOS обнаруживает, что символ control-s присутствует на консольном вводе во время консольного вывода, все намертво остановится до тех пор, пока с консоли не будет введен другой символ, и только после этого управление будет возвращено из функции BDOS. Все будет хорошо, если вы хотите иметь возможность остановить и продолжить вывод без необходимости кодировать эту особенность в вашей программы с, но это вызывает большие неприятности, если Вам необходимо получать каждый символ, введенный с консоли, включая control-s. Так как BDOS должна иметь возможность обнаружить control-s на консольном вводе, она должна читать с консоли в любой момент, когда она видит, что был введен символ. Если введен символ, не требующий такой специальной обработки, как для control-s, тогда он должен быть сохранен где-то внутри BDOS, таким образом следующий вызов "консольный ввод" вернет этот символ, как будто ничего не случилось. Кроме того, BDOS должна на любые последующие вызовы функции "получить консольный статус" (Get control status) отвечать, что во вводе имеются символы. Это приводит к ситуации, при которой BDOS может сказать, что символ имеется, тогда как соответствующий вызов BIOS не выдал бы подобного ответа.

Если все это вам показалось непонятным, знайте, что автору потребовалось провести наедине с CP/M и первыми версиями компилятора несколько долгих месяцев, прежде, чем он смог понять, что происходит. Библиотечные версии GetChar и PutChar были созданы для тех условий, в которых пользователь не нуждается в абсолютном управлении консолью. Так как BDOS уже делает очень приятные операции (например, обработку control-s), автор добавил еще несколько операций: автоматическое преобразование символа '\n' в комбинацию cr-lf на выходе, автоматический останов выполняемой программы, когда обнаруживается control-c при вводе или выводе (поэтому программы с бесконечным выводом могут быть прерваны без перезапуска машины, даже, когда не выполняются никакие операции консольного ввода), автоматическое преобразование символа возврата каретки в '\n' и т.д. один неопытный пользователь заметил, что он хотел бы, чтобы PutChar была освобождена от реакции на control-c; для него автор добавил библиотечную функцию PctCh, которая действует так же, как PutChar, за исключением того, что control-c не останавливает программу при выводе на консоль. Немного позже станет очевидным, что ни PutChar, ни PctCh не подходят для случая, когда требется предохранить программу от прерывания вывода. Тогда автор и добавил функцию BIOS, так что пользователи могут делать ввод/вывод прямо через BIOS и полностью отказаться от BDOS.

Автор обещал несколько примеров, давайте рассмотрим их.

Прежде всего, разберем самый элементарный набор функций без каких-то специальных преобразований или перехватов т.е., никаких переводов типа '\n' в cr-lf.

/\* ультранеобработанные консольные функции ввода/вывода

\*/

GetChar() /\* получите знак с консоли \*/

{

return BIOS (3);

}

KbHit() /\* верните значение True, если на вводе го-

тов символ \*/

{

return BIOS (2);

}

PutChar(c) /\* запишите знак с на консоль \*/

char c;

{

BIOS (4,с);

}

Эти ультранеобработанные функции ничего не делают, кроме обеспечения доступа к консольным подпрограммам BIOS. Чтобы использовать их вместо стандартных версий, данных в deff2.crl (которые случайно записаны на языке ассемблера и имеются в исходной форме в deff2a.csm), просто включите их в исходный файл, откомпилируйте и откомпонуйте его и в дальнейшем компонуйте ваши программы с полученным crl-файлом.

Рассмотрим еще несколько примеров. Для начала давайте создадим набор функций, которые выполняли бы преобразования символа "новая строка", как библиотечные версии, описанные ранее, осуществляли бы выход при вводе control-c, но игнорировали бы протокол control-s/control-q и отбрасывали все символы, введенные пользователем во время вывода, за исключением control-c, который должен привести к возврату на командный уровень. Таким образом, нужно расширить предыдущие функции до следующих возможностей:

а) преобразование символов '\n' в два символа cr и lf при выводе;

b) преобразование <вк> в "новую строку" ('\n') и control-z в -1 при вводе;

с) эхо-вывод при вводе;

d) выход в систему при вводе control-c во время ввода и

вывода.

Вот новые версии:

/\* тепленькие консольные функции ввода/вывода без использования BDOS: следует заметить, что KbHit будет той же самой, что и в предыдущей ультранеобработанной версии) \*/

# define CtrlC 0x03 /\* control-c \*/

# define CPMEOF 0x1a /\* сигнал конца файла \*/

GetChar () /\* получите знак, новая версия \*/

{

char c;

if ((c=BIOS(3)) == CtrlC) BIOS(1);

if (c==CPMEOF) return -1; /\* превратим ctl-z в -1 \*/

if (c=='\r')

{

/\* если cr введено, to \*/

PutChar ('\r'); /\* эхо-вывод cr и затем \*/

c='\n'; /\* сделаем так, чтобы при \*/

} /\* общем эхо-выводе был выведен lf \*/

/\* и произошел возврат '/n' \*/

PutChar (c); /\* общий эхо-вывод \*/

return c; /\* и возвращение символа \*/

}

PutChar (c) /\* вывод символа , новая версия \*/

char c;

{

BIOS (4,c) /\* прежде выведем символ \*/

if (c == '\n') /\* если это новая строчка \*/

BIOS (4,'\r'); /\* то выведем также cr \*/

if (KbHit()&&BIOS(3) == CtrlC) /\* если введен ctrl-c \*/

BIOS (1) /\* выход \*/

}

Теперь, если вы хотите добавить обработку символа control-s и свойство обработки незапланированного ввода (чтение с забеганием вперед) (оба довольно взаимосвязаны, так как вы должны иметь возможность сохранять во вводе все, за исключением control-s, что могло быть обнаружено во время вывода), вы можете завести некоторые внешние переменные, отображающие "состояние" для этих функций, и с их помощью отслеживать консольный ввод. Как только это будет сделано, вы будете иметь те же функции, что и стандартные библиотечные версии GetChar и PutChar (которые используют BDOS).

К большому сожалению автора, имеется консольная операция, которая часто нужна при программировании интерактивных операций реального времени, не поддерживаемая BIOS и таким образом, нет мобильного способа реализовать ее в CP/M. Это функция, которая могла бы говорить, готов ли консольный терминал принять символ для вывода. Программа должна быть способна считывать ввод с клавиатуры в любой момент и не может позволить себе простаивать в ожидании готовности вывода, если объем выводимых данных вызвал приостановку программы до тех пор, пока символ не сможет быть послан. Единственный способ послать символ на консоль - при помощи вызова ConOut BIOS и такой вызов может в любое время приостановить программу на более длительный срок, чем допустимо. Выход из положения - отказаться от ConOut и создать свою программу вывода, которая будет более сложной. Каждый пользователь должен индивидуально создавать свои файлы заголовков, чтобы определить уникальные номера портов, позиции битов и полярности ввода/вывода технических средств, управляющие его консолью. Было бы гораздо легче, если бы BIOS содержала еще одну подпрограмму для тестирования статуса вывода на консоль.

**51. Библиотека функций cio.crl.**

Новый пакет cio (в исходной форме cio.csm) был включен в BDS C версия 1.50 для использования в тех случаях, где требуется полное прямое управление консольным вводом/выводом.

**52. Пакет обработки чисел с плавающей точкой.**

Компоненты пакета для выполнения операции с плавающей точкой:

1) float.c - файл с текстами функций, написанных на с

2) Fp - рабочая функция (в deff2.crl)

3) floatsum.c - демонстрационный пример использования пакета

Вот как пакет работает: для каждого числа с плавающей точкой, с котоpым вы хотите работать, вы должны объявить массив из пяти char (байт). Затем передайте указатель на массив соответствующей функции. Ее аргументы являются указателями на такие массивы. Четыре основные арифметические функции: FpAdd, FpSub, FpMul и FpDiv. Каждая из них принимает три аргумента: указатель на массив из пяти байтов, где будет результат, и два операнда (каждый указатель на массив из пяти байтов, представляющий операнд с плавающей точкой).

Следует заметить, что результат может быть помещен в один из операндов без фатальных последствий, т.е. операция:

FpMult (foo, foo, foo) успешно возведет в квадрат foo и поместит результат в foo.

Чтобы инициировать массивы для чисел с плавающей точкой, а также печатать значения в читабельной фоpме, имеются следующие функции:

FtoA - преобразует число с плавающей точкой в цепочку ASC II, (котоpую вы можете затем вывести на экран при помощи "PutS"). Следует заметить: эта функция не нужна при использовании специального средства PrintF в float.c.

AtoF преобразует цепочку ASC II (терминированная нулем) в число с плавающей точкой.

ItoF преобразует целое число в число с плавающей точкой.

**53. Краткое описание функций.**

Следующие функции позволяют пользователям BDS C иметь доступ к вещественным числам и манипулировать ими. Каждое вещественное число должно быть размещено в массиве из пяти байт (пример: char Fpno[5]). Первые четыре байта содержат мантиссу с младшим первым байтом, пятый байт является поpядком числа.

FpComp (Op1, Op2)

char Op1[5], Op2[5];

возвращает:

1 , если Op1 > Op2

-1 , если Op1 > Op2

0 , если Op1 == Op2

Как и в большинстве других пакетов с плавающей арифметикой нежелательно проверять вещественные числа на равенство.

char \*FpAdd (Result, Op1, Op2)

char Result[5], Op1[5], Op2[5];

Записывает значение Op1+Op2 в Result. Op1 и Op2 должны быть числами с плавающей точкой. Возвращает указатель на начало результата.

char \*FpSub (Result, Op1, Op2)

char Result[5], Op1[5],Op2[5];

Записывает значение Op1-Op2 в Result. Op1 и Op2 должны быть числами с плавающей точкой. Возвращает указатель на начало Result.

char \*FpMult (Result, Op1, Op2)

char Result[5], Op1[5], Op2[5];

Записывает значение Op1\*Op2 в Result. Op1 и Op2 должны быть числами с плавающей точкой. Возвращает указатель на начало Result.

char \*FpDiv (Result, Op1, Op2)

char Result[5], Op1[5], Op2[5];

Записывает значение Op1/Op2 в Result. Op1 и Op2 должны быть числами с плавающей точкой. Возвращает указатель на начало Result. при делении на нуль возвращает нуль.

char \*AtoF (Op1,s1)

char Op1[5], \*s1;

Преобразует цепочку ASC II s1 в число с плавающей точкой и заносит результат в Op1. Функция будет игнорировать лидирующие пробелы, но внутри строки пробелы недопустимы. Примеры допустимых строк:

"2", "22022222222383.333", "2.71828е-9",

"334.3333е32".

Строка такая, как "3443.33 е10" недопустима, т.к. содержит внутри себя пробел. Значение показателя должно быть задано в пределах:

-38<= показатель <=38.

Возвращается указатель на результат.

char \*FtoA (s1, Op1)

char \*s1, Op1[5];

Преобразует число с плавающей точкой Op1 в строку ASC II s1. Оно будет фоpматировано в scientific-нотации, то есть в виде "+9.9999999е+38" с точностью семь (7) цифр после точки.

Строка будет завершена нулем.

Возвращает указатель на начало s1.

char \*ItoF (Op1, n)

char Op1[5];

int n;

Присваивает числу с плавающей точкой Op1 значение целого числа n. N считается целым числом со знаком.

**54. Общие вопросы.**

Операции с плавающей точкой должны рассматриваться, как вызовы функций, а не как простые встроенные подвыражения; этому необходимо уделить особое внимание, чтобы не перепутать возможности компилятора с возможностями пакета с плавающей точкой. Чтобы присвоить числу с плавающей точкой начальное значение, например, вы можете машинально указать:

char Fpno [5];

Fpno="2.236";

Однако для того, чтобы добиться желаемого результата, Вы должны указать:

char Fpno[5];

AtoF(Fpno,"2.236");

Далее, пусть вы хотите присвоить числу с плавающей точкой значение целой переменной IVal. При этом последовательность операторов:

char Fpno[5];

int IVal;

...

Fpno=IVal

не будет работать, вы должны изменить последнюю строку на

ItoF (Fpno, IVal);

Еще несколько примеров:

сложим числа 100.2 и -7.99, :

FpAdd (a, AtoF(b,"100.2"), AtoF(c,"-7.99"));

(следует заметить, что a, b и c должны быть массивами из пяти байт).

Следующий оператоp не прибавит 1 к а, так как Op1 и Op2 должны быть числами с плавающей точкой (т.e. указатели на байты):

FpAdd (a,a,1); /\* плохое использование "FpAdd" \*/

Все выше упомянутые функции записаны на Си, но большинство из них вызывает одну рабочую функцию Fp, котоpая и делает всю работу. Эта функция помещена в deff2.crl; это единственная закодированная на машинном языке часть пакета.

Внимание: набор функций плавающей арифметики для пользователей Orionsoft будет поставляться отдельно (вместе с другими наборами) отдельным пакетом.

**55. Длинные целые числа.**

Этот пакет добавляет к BDS C возможности обработки длинных (32 бита) чисел со знаком в том же духе, что и в пакете плавающей арифметики. Обеспечивается сложение, вычитание, умножение, деление и взятие остатка от деления, а также сравнение, присваивание и разные виды преобразований.

Каждое длинное целое число хранится как массив из 4-х байтов (char) число х, таким образом, определяется, как:

char x[4];

Внутреннее представление - это форма дополнительного кода.

Старший байт - первый в массиве. Однако, для ваших целей вам не нужно знать внутреннее представление.

Большинство программ, которые работают с длинным целым числом имеет три аргумента, первый из которых указывает, где будет храниться результат, а другие два - операнды. Например, пусть имеются длинные целые числа x, y, z (все определены как char[4]), при этом оператор:

LAdd (z,x,y);

Подсчитывает сумму x и y и заносит ее в z, указатель на z и возвращается. Следует заметить, что аргумент-результат может совпадать с одним или с обоими операндами. Например, LAdd

(x,x,x) делает ровно то, что имеется ввиду.

Пакет был написан частично на Си, а частично на языке ассемблера 8080 (для скорости и компактности). Чтобы использовать его, просто компонуйте long.crl с вашей программой.

Ниже дано описание каждой функции.

ItoL (l,i)

char l[4];

int i;

Заносит длинное представление 16-битового числа i в l и возвращает l.

AtoL (l,s)

char l[4];

char \*s;

Заносит длинное представление строки Asc II s в l и возвращает l.

Формат строки s - последовательность десятичных цифр, перед ней может стоять знак минус, завершается строка произвольным символом, не являющимся цифрой.

LtоA (s,l)

char \*s;

char l[4];

Заносит символьное представление длинного целого числа l в строку s и возвращает s. Представление состоит из строки цифр Asc II, перед которой стоит знак минус, если l отрицательное длинное целое число. Длина s должна быть достаточной, чтобы разместить результат.

LAdd (r,Op1,op2)

char r[4];

Заносит сумму длинных целых чисел Op1, op2, в r и возвращает r. Op1 или Op2 могут использоваться в качестве r.

LSub (r,op1,op2)

char r[4];

char Op1[4], op2[4];

Подобно LAdd, но считает оp1 - op2.

LMul (r,Op1,op2)

char r[4];

char Op1[4], op2[4];

Подобно LAdd, но вычисляет оp1 \* op2.

LDiv (r,Op1,op2)

char r[4];

char Op1[4], op2[4];

Подобно LAdd, но вычисляет численное отношение оp1/op2. Если оp2 - нуль, результат равен нулю.

LMod (r,Op1,op2)

char r[4];

char Op1[4], op2[4];

Подобно LAdd, но вычисляет Op1 Mod op2. Если ор2 - нуль, в

результате получится нуль.

LComp (Op1,op2)

char Op1[4], op2[4];

Сравнивает длинные целые числа Op1 и ор2 и возвращает одно из значений 1,0,-1, в зависимости от результата сравнения (Op1>ор2), (Op1=ор2) или (Op1<ор2).

LAsSign (dest, source)

char source[4], dest[4];

Присваивает длинному целому числу desт значение длинного

целого числа source и возвращает указатель на dest.

ltou (l)

char l[4];

Преобразует длинное целое число l в число без знака (при помощи усечения).

utol (l,u)

char l[4];

unsigned u;

Заносит длинное представление беззнакового целого u в l и возвращает l.

**56. Детали реализации.**

Большая часть работы в выше описанных функциях делается одной функцией на языке ассемблера, называемой Long, исходный текст которой имеется в файле long.csm (доступен для членов user's group bds). Остальная часть располагается в файле long.c. следует заметить, что большинство примитивов, описанных выше, просто вызывают long, передавая ей код операции (который говорит, какая операция должна выполняться) вместе с аргументами.

Файл long.crl содержит компилированные функции, заданные в long.c, а deff2.crl содержит рабочую функцию long.

Внимание: данный набор функций будет поставляться (вместе с другими наборами) в отдельном пакете.

**57. Отладчик для BDS C.**

Это приложение является основной документацией для отладочного пакета CDB. Представленная информация должна быть достаточной, чтобы вы смогли решить, нужно ли вам доставать пакет CDB.

**58. Объяснение компонентов.**

CDB - это интерактивный символический отладчик для программ, написанных для компилятора BDS C.

CDB позволяет пользователю вводить точки останова в программе, трассировать ход выполнения программы, выводить на дисплей и изменять переменные, указывая их имена. Таким образом, разработчик обеспечивается средством, которое (как надеется автор) будет полезным для разработки и тестирования программ.

Пакет отладки состоит из трех выполняемых файлов. Первый (l2.com) является компоновщиком для объектных файлов в формате "crl" и является заменой стандартной программы clink, поставляемой с BDS C. L2 в нашем отладочном пакете - это слегка модернизированная версия редактора связи clink, он объединяет все возможности clink со средствами отладки и индикации ошибок. L2 подготавливает загрузку и выполнение "com"-файла под управлением других частей пакета отладки, a также готовит таблицу символов для использования пакетом.

Второй элемент пакета - cdb.com - используется разработчиком программы (который в данном документе называется "пользователем"), чтобы активизировать отладчик. CDB интерпретирует аргументы командной строки, введенной пользователем, готовит различные таблицы и активизирует cdb2.ovl, последний элемент пакета. Cdb2 размещается в памяти под BDOS, загружает программу, которую нужно отладить ("целевая программа") в начало TPA с адреса 100h.

Cdb2 остается резидентным в памяти вместе с отлаживаемой программой в течение всего сеанса отладки. После того, как cdb2 загрузила программу, она передает управление отлаживаемой программе, которая начинает выполняться. Когда отлаживаемая программа:

1) выполняет вызов функции

2) возвращает управление из функции

3) встречает начало кода для какого-либо оператора Си, управление передается в cdb2, который либо возвращает управление обратно, либо останавливает выполнение и ожидает ввод пользователем команд отладчика.

Примечание: предыдущее утверждение нельзя понимать буквально, не каждое наступление одного из перечисленных событий вызывает активизацию cdb2. Смотрите пояснение ниже по поводу системных библиотек и опции -s и -ns программы l2.)

В этом документе квадратные скобки "[]" используются, чтобы обозначить необязательные элементы, т.е. элементы, которые могут быть опущены.

**59. Настройка и сборка отладчика.**

Из-за большого количества различных изменений, которые необходимо сделать в коде нескольких компонентов отладчика, пакет поставляется в исходном виде. Эта часть документов описывает шаги, которые необходимо сделать, чтобы трансформировать исходный код в три файла l2.com, cdb.com и cdb2.ovl.

**60. Куда поместить cdb2.**

До построения cdb.com или cdb2.ovl прежде всего необходимо решить, где cdb2.ovl будет размещаться в памяти. cdb2 размещается со старших адресов памяти над целевой программой и ее стеком, но ниже BDOS CP/M и собственного стека cdb2. Код, который составляет cdb2 - это немного меньше по длине, чем 0x4600 байтов (т.е. 18 кбайт). Внешние данные занимают около 0x0480 байтов. Автор надеется, что этого достаточно, но т.к. можно сотворить достаточно сложные выражения, которые должны быть синтаксически разобраны (рекурсивно), чтобы дампировать содержимое переменных, автор позволяет себе напомнить, что краткость - это лучшее качество.

Складывая эти величины, мы получим в итоге 0x5400 байтов для кода, внешних данных и стека для cdb2; таким образом, cdb2 должна располагаться на 0x5400 байтов ниже начала BDOS. Т.к. BDOS автора начинается с адреса 0xe406, то его cdb2 начинается с ячейки 0x9000 (и автор использует это значение в примерах ниже). Если вы не знаете адрес вашего BDOS, самый простой способ выяснить его - использовать ddt, чтобы рассмотреть адресное поле команды перехода в ячейке 0005. Чтобы это сделать, прежде всего введите "ddt". После вывода подсказки ddt, введите "l5". Ваша BDOS расположена в памяти, начиная с адреса, который стоит в первой строке ответа ddt.

Дистрибутивный диск содержит версию cdb.com и cdb2.ovl, установленную для системы с BDOS, размещаемой не ниже d300. Почти все системы с ram более 60к должны использовать эту версию, как есть. Однако, эта версия оставляет только 31к для отлаживаемой программы и символьных таблиц; если ваша система имеет BDOS с началом выше чем d300, вы можете пожелать, чтобы отладчик дал вам больше памяти для отлаживаемой программы; и, если ваша система имеет BDOS ниже 0xd300, вы обязаны преобразовать CDB и получить работающий отладчик.

Если вы уже решили, где помещать cdb2, вы должны отредактировать CDB и изменить #define для cdb2addr на то значение, которые вы уже определили. Ниже автор использует "cdb2addr", чтобы сослаться на это значение. редактируя cdb.h, вы можете изменить #define для cdb2-drive; это специфирует дисковод, из которого cdb.ovl будет загружена, если пользователь не отвергает умолчание, используя опцию "-d" в командной строке CDB. Вы можете специфицировать либо обозначение диска (без двоеточия), например, "A", либо букву диска с префиксом номера пользователя, например, "0/а". В дистрибутивном варианте предполагается по умолчанию отсутствие обозначения диска, что означает загрузку cdb2 из текущего зарегистрированного диска и области пользователя.

**61. Построение CDB.**

После изменения cdb2addr в cdb.h вы готовы компилировать два исходных файла для CDB:

cc cdb.c -c 3200 cc build.c -c 3200 l2 CDB build

Имеется submit-файл - cdb.sub для выполнения выше представленной последовательности команд.

**62. Построение cdb2.**

Чтобы оттранслировать исходные файлы для cdb2, нам необходимо знать адрес внешних данных cdb2. Т.к. внешние данные помещаются сразу после кода cdb2, мы просто прибавляем 0x4600 (размер кода, заданного выше в разделе в) к cbd2addr. В данном случае результатом будет 0xd600; таким образом, чтобы компилировать cdb2 для данной системы автор специфицирует опцию

"-ed600".

Cdb2 состоит из семи исходных файлов; чтобы компилировать их, вы можете ввести:

cc cdb2.c - exxxx

cc atbreak.c - exxxx

cc break.c - exxxx

cc command.c - exxxx

cc print.c - exxxx

cc parse.c - exxxx

cc util.c - exxxx,

где xxxx-должны быть заменены адресом внешних данных (например, 0xd600) или использовать submit-файл cdb.sub. Чтобы использовать cdb2.sub, просто введите:

submit cdb2 xxxx,

где xxxx является местом размещения внешних данных. Или задайте:

submit cdb2 xxxx d:,

где d является обозначением дисковода, на котором размещаются исходные файлы, если они не на дисководе по умолчанию.

Как только файлы откомпилированы, вам необходимо ассемблировать один исходный файл dasm.csm, (см. приложение, касающееся casm). Введите

casm dasm

asm dasm

ddt dasm.hex

g0

save 3 dasm.crl

Чтобы спасти вас от этого неудобства, особенно, если у вас под рукой нет компилированной версии casm, дистрибутивный диск содержит предварительно ассемблированный dasm.crl. Последний файл, который нужно создать - это пустой файл, называемый null.sym, который l2 будет пытаться использовать для определения адресов всех функций, используемых в корневом сегменте, для которого cdb2.ovl будет оверлейным сегментом. Так как нет такого корневого сегмента, нет также функций; но l2 требует имя корневого сегмента ,если используется опция -ovl, поэтому мы создаем пустой файл, чтобы угодить компоновщику, выдавая:

save 0 null.sym

Теперь, когда все crl-файлы готовы, мы можем откомпоновать их. Правильная команда

l2 cdb2 dasm atbreak command break print parse util -ovl

null yyyy -wa,

где yyyy должно быть заменено на значение, подсчитанное для cdb2addr в шестнадцатиричном виде. Имеется submit-файл lcdb2.sub для выполнения выше указанной процедуры. Теперь отладчик готов к использованию.

**63. Изменение номера рестарта.**

В поставляемом варианте пакет отладчика использует команду rst 6, чтобы генерировать точки останова. Всякий раз, когда встречается команда rst 6, управление передается в ячейку 0x0030. В некоторых системах эта ячейка памяти (или сама команда rst 6) может быть предназначена для использования в других целях. Если это так, то есть необходимость назначить какой-то другой номер рестарта для функции точки останова. Любой номер рестарта от 1 до 7 (включительно) может быть использован; не разрешен рестарт 0. Чтобы изменить номер рестарта, необходимо внести изменения в l2.c, cdb.h и dasm.csh.

И в l2.c и в cdb.h #define для rstnum должно быть изменено на номер рестарта, который пользователь предназначил для отладчика. В dasm.csm "equ" для rstnum должна быть заменена на то же самое значение. Следует заметить, что значение должно бы специфицировано, как число от 1 до 7.

В заключение, заметим, что, когда целевая программа будет компилироваться (с опцией "-к"), необходимо специфицировать новый номер рестарта, вместо обычного "-к" введите "-кн", где н номер рестарта.

**64. Как активизировать отладчик.**

Чтобы использовать отладчик, пользователь должен прежде всего компилировать и компоновать целевую программу и затем активизировать сам отладчик. В данной главе описывается этот процесс. Для лучшего понимания частей 3 и 4 этого документа, в части 6 приведен пример сеанса отладки.

Компоновка: опция "-k"

Как указано в руководстве пользователя BDS C, опция "-к" используется, чтобы заставить компилятор генерировать символьную таблицу с расширением ".cdb" и генерировать команды рестарта в компилированном коде. Пользователь вводит команду "cc", как для обычной компиляции, но добавляет опцию "-к". Например:

cc target.c -k

Компоновка: опции l2 -d, -s, -ns.

Чтобы компоновать отлаживаемую программу, пользователь должен использовать компоновщик l2 вместо clink. Как описано в документации, l2 имеет отличный от clink синтаксис командной строки; кроме того, отладочная версия l2 имеет следующие дополнительные опции:

-d создает выходной модуль, который совместим с CDB. Эта опция заставляет l2 поместить команду рестарта в начале большинства функций. Если не задана опция -s или -ns, команда рестарта помещается в начало каждой функции, за исключением тех функций из deff\*.crl, к которым обращаются из функций, которые сами входят в deff\*.crl.

-s: crl-файлы после -s будут рассматриваться как "системные" библиотечные файлы. Функция в системном библиотечном файле, к которой можно обратиться только из функции из системного библиотечного файла, не будет иметь в начале себя команды рестарта, добавленной l2, и отладчик не будет трассировать вызов такой функции. Опция -d без -s или -ns подобна "-s deff deff2 deff3". "-ns" специфицирует, что не должно быть никаких системных библиотечных файлов; эта опция используется, чтобы отвергнуть умолчание, согласно которому файлы deff\*.crl являются системными библиотеками.

Пример:

l2 target -d

**65. Активизация CDB.**

Чтобы активизировать отладчик, пользователь вводит команду CDB. Командная строка имеет следующий вид:

CDB <имя целевой программы> [-l[local-cdbs]]

[-g[globalcdbs]]

[-d [user/]drive] [%[операнды целевой программы]] <вк>

-l и -g позволяют пользователю специфицировать ".cdb"-файлы, из которых CDB будет читать таблицы символов, содержащие информацию о переменных, используемых в отлаживаемой программе. используется файл "имя-целевой-программы".cdb, если l или g не специфицированы; хотя это умолчание обычно верно, однако, если исходный код содержится в более, чем одном файле, пользователь должен обеспечить имена CDB-файлов, полученных из каждого исходного файла, если он желает иметь доступ к символам, определённым в этих файлах. Часто все глобальные данные определяются в заголовочном файле (с расширением .h), который включен в каждый исходный файл; в таком случае нет необходимости использовать опцию -g, только -l. С обеими этими опциями, если пользователь выводит нули вместо имени файла, CDB не будет загружать символьные файлы с данным типом символов (локальные или глобальные); если пользователь не вводит вообще никакого аргумента для этих опций, CDB попросит пользователя ввести имена файлов, по одному в строке. Пустая строка завершает ввод.

Операнд "%" позволяет пользователю специфицировать аргументы, передаваемые в отлаживаемую программу. Если за "%" следуют операнды, эти дополнительные операнды будут переданы прямо в отлаживаемую программу; если ничего не следует за "%", будет введена подсказка для ввода командной строки.

Примечание для любителей покопаться внутри: CDB не передает аргументы, которые следуют за "%", через переданный CDB "ArgV"; напротив, CDB изменяет аргументы, как они появляются в памяти в 0x0080 и позволяет целевой программе через c.ccc анализировать эту командную строчку.

Опция -d специфицирует дисковод (с необязательным цифровым префиксом пользователя), из которого cdb2.ovl будет загружен; по умолчанию - текущий дисковод, но пользователь может отвергнуть умолчание. Стандартный вызов:

A>cdb cdbtarget

**66. Резюме.**

Стандартная процедура для отладки программы Си именем tar-get.c выглядит следующим образом:

cc target.c -k

l2 target -d

cdb target

Для более сложного примера примем, что foo.c содержит исходный текст функции "Main" и других функций, и что bar.c и lib.c содержат исходные тексты для других необходимых функций, а кроме того, одинаковы описания для внешних переменных (оба исходные файла содержат #include "globails.h"), в то время, как lib.c содержит библиотечные функции пользователя, которые не имеют доступ к общим переменным. наконец, допустим, что пользователь имеет определенные (уже отлаженные) функции в stdlib.crl. Чтобы компилировать эту смесь, пользователь вводит:

cc fоо.c -k

cc bar.c -k

cc lib.c -k

Чтобы скомпоновать все вместе для получения fоо.com, пользователь вводит:

l2 fоо bar -l lib -s stdlib

Операнд -s предписывает l2 не генерировать отслеживание обращений к программам, включенным в fоо.com, которые были вызваны только программами в stdlib.crl. Чтобы активизировать отладчик, пользователь вводит:

cdb fоо -l bar lib

Операнд -l говорит CDB, что файлы bar.cdb и lib.cdb содержат таблицу символов, выданную cc и что информация обо всех локальных символах из этих файлов должна быть загружена. Загружается информация обо всех как локальных, так и глобальных

символах из foo.cdb.

**67. Команды отладки.**

В этой части документации обсуждаются различные команды CDB, сгруппированные по функциям.

Когда активизируется отладчик, он выводит на дисплей адрес начала cdb2 (t.e.cdb2addr), объем пространства, занятого локальными и общими таблицами символов и вершину стека отлаживаемой программы (т.е. самый старший байт, не занятый cdb2 и ее таблицами). Отладчик затем передает управление отлаживаемой программе, которая после выполнения инициализации c.ccc, активизирует функцию "Main" целевой программы. т.к. точка останова устанавливается на входе в "Main", управление затем передается обратно пользователю, которому выдается запрос на ввод команды.

**68. Точки останова.**

CDB обычно разрешает отлаживаемой программе выполнять один оператор за другим без прерывания. У пользователя есть два способа, при помощи которых он может приостановить выполнение: задать точку останова и прервать выполнение с клавиатуры. Назначая точки останова, пользователь заставляет CDB остановить выполнение по завершении текущего Си-оператора. чтобы генерировать прерывание, пользователь просто вводит любой символ; когда CDB воспринимает этот символ, он останавливает выполнение (однако следует заметить, что, если отлаживаемая программа ждет ввода символа пользователем, он будет ей передан и не вызовет прерывания). Чтобы установить точку останова, пользователь вводит команду "break": b[reak] [имя-функции] [номер-оператора[count]] символы, заключенные в скобки, могут быть опущены; таким образом, команда "break" может быть введена как "b", "br", "bre" и т.д. Имя функции и номер оператора могут быть опущены. Если имя функции опущено, точка останова устанавливается на заданном операторе текущей функции, т.е. функции, которая сейчас отлаживается. Имя этой функции выдается сdb на экран, когда выполнение отлаживаемой программы остановлено и может быть распечатано командой "list". Номер оператора явно указывает, где в специфицированной функции нужно назначить точку останова. операторы нумеруются по строкам, первая строка функции имеет номер1. Если несколько операторов находятся в одной строке, например:

A=5; PutChar("x"); while (\*s) s++;

используется "дробная" нотация; первое утверждение в строке n нумеруется "n.0", следующее "n.1" и т.д. (итак в нашем примере, принимая что данная строчка является 5-ой строчкой в функции, "a=5;" имеет номер 5.0; "putchar(ЪxЪ);" - 5,1; "while(\*s)"-5.2 и "s++"-5.3. Когда не задано никакого числа, принимается ".0". Таким образом номер оператора может быть определен как:

sn:= <номер строки> [.номер оператора в строке]

В более сложном случае cc реорганизует исходный код или генерирует свои собственные операторы. Когда это случается, для пользователя труднее назначать точку останова на желаемом операторе. Самыми важными случаями, в которых cc генерирует эти

"спрятанные операторы" является: в циклической конструкции ("while","for","do"), компилятор генерирует команды ветвления из конца цикла в его начало; в операторе "for" cc перемещает "инкрементную часть" (т.е. последний из трех элементарных операторов, заключенных в операторе "for") в конец цикла; таким образом, этот оператор не считается вместе с остатком оператора "for", а следует за последней строкой цикла.

Помимо вышеперечисленных, имеются два специальные номера для оператора - это 0 и -1. 0 - это вход в функцию и точка останова располагается перед кодом выполняемой функции. -1 - это возврат из функции; точка останова располагается после выполнения возврата (и, таким образом, возвращенное функцией значение можно вывести на дисплей). Точки останова могут быть установлены на "операторы" 0 и -1 так же, как и на любые другие операторы. До сих пор совсем не упоминался операнд count. Назначение точки останова командой "break" заставляет CDB не останавливать выполнение целевой программы до тех пор, пока точка разрыва не встретится ровно count раз. По умолчанию - 1. Одновременно может быть назначено до 40 точек останова. команда "reset" используется для того, чтобы убрать точку останова. Синтаксис ее следующий: r[eset] [имя-функции] [номер оператора] значения по умолчанию те же, что и для команды "break". Конечно, возникнет ошибка при попытке удалить не назначенную точку останова. Команда "clear" может использоваться, чтобы удалить все точки останова. Синтаксис: clear команда "clear" должна быть введена полностью. Команда "list breakpoints" может использоваться, чтобы выдать листинг всех назначенных в настоящее время точек останова.

**69. Выполнение программы.**

Имеется несколько команд, которые используются для выполнения скомпилированного кода. Первая из команд - это "go", она просто запускает выполнение (с того места, где оно было последний раз остановлено) и продолжает его до тех пор, пока не встретится точка останова или пользователь не нажмет какую ни будь клавишу. команда не имеет никаких операндов. Чтобы увидеть, какие операторы выполняются программой на выходном языке, пользователь может использовать команду "trace". Команда:

t[race] [число операторов]

заставляет отладчик выполнять указанное количество операторов, каждый раз печатается имя функции и номер выполняемого оператора. Выполнение завершается после того, как заданное число операторов выполнено, или при достижении точки останова, или при прерывании с клавиатуры. По умолчанию число операторов равно 1. "untrace" (синоним "walk") подобна команде "trace", за исключением того, что имена функций и номера операторов не выводятся на дисплей. Другими словами:

u[ntrace] [число операторов]

Заставляет отладчик выполнять указанное число операторов. Как и для "trace", выполнение завершается после того, как будет отработано требуемое число операторов, когда встречается точка останова или при прерывании с клавиатуры. Значение по умолчанию для числа операторов равно 1.

И последний способ выполнения - это команда "run" (слово "run" нельзя сократить. Этот оператор заставляет cdb передать управление отлаживаемой программе и совершенно дезактивирует отладчик; как только введен "run", возврат в отладчик уже невозможен.

**70. Дампирование переменных.**

Команда "dump" используется, чтобы вывести содержимое области памяти. Синтаксис команды:

d[ump] выражение [множитель] [формат]

синонимы для "dump" -"p[rint]"и "," (запятая).

Команда "dump" дампирует область памяти, начало которой задается "выражением". Хотя полное определение "выражения" дано ниже, две наиболее общие формы - это имя переменной (такое, как "i","f00" или "filename") и целое число в шестнадцатеричном или десятичном формате (таком, как 0x0100,43000 или 12). Если задано имя переменной или другое символическое выражение, cdb будет дампировать переменную в формате, соответствующем описанию этой переменной; если переменная является структурой, cdb будет дампировать каждый элемент структуры. Однако пользователь может специфицировать другой формат для использования, и часто вводит такую спецификацию, когда выражение не является символическим, а является адресом. Допустимы форматы:

c - символ

p - указатель

i или w - целое число/слово

s - строка, завершаемая нулем ('\0') является форматом по умолчанию, если не специфицирован никакой формат для не символического выражения. Операнд "множитель" специфицирует, сколько необходимо дампировать памяти. Команда "dump" дампирует соответствующее число экземпляров специфицированного формата; таким образом:

dump 0x0100 10c

будет дампировать десять байтов (символов) от 0x0100 до 0x010a, в то время, как

dump 0x0100 10

будет дампировать десять слов (двадцать байтов), от 0x0100 до 0x0114, т.к. "w" является форматом по умолчанию. Синтаксис для выражения следующий:

<выражение>::=<выражение первичное>

<первичное>::=<целое>

имя

(выражение)

первичное[выражение]

первичное.имя

первичное->имя

Это, вообще говоря, означает, что любое выражение, которое не содержит логической или арифметической операции, является допустимым выражением для cdb; выражения могут быть достаточно сложными, например:

table [table [1,i],j].name [10]

Чтобы остановить чрезмерно длинную выдачу команды "dump", вводят любой символ.

Обычно правила идентификации Си используются для символических ссылок. Это означает, что, когда отладчик встречает точку останова в функции "fоо", ссылка на переменную "bar" указывает на переменную, локальную для "fоо" с именем "bar", если такая переменная существует, если такой локальной переменной нет, ссылка осуществляется на глобальный символ "bar". Это правило идентификации делает невозможным доступ к глобальной переменной из функции, имеющей локальную переменную с тем же именем, что и глобальная переменная. Cdb позволяет пользователю отвергнуть стандартное правило действия имен и специфицировать глобальную переменную при помощи обратного слеша ("\"). В выше приведенном примере, чтобы иметь доступ к глобальной переменной "bar" из функции "fоо", пользователь может ввести:

dump \bar

Последний вариант использования команды "dump" - это нахождение адреса, а не значения переменной. Для этого "выражение" прификсируются амперсандом. Например, чтобы определить адрес переменной по имени table, введите:

dump &table

сложное символическое выражение может также быть использовано:

dump &table [ij]

Установка значений - команда "set" используется, чтобы занести данные в память. Синтаксис:

s[et] выражение значение [c]

Запишет значение в ячейку памяти, к которой можно обратиться при помощи "выражения". Обычно записывается 16-битовое значение; однако, если выражение является символическим выражением, которое обозначает переменную типа char или значение заключено в одинарные кавычки, например, '#' или указана опция "c", то записывается только 8-битовое значение.

**71. Команды распечатки.**

Команда "list" используется, чтобы иметь доступ к различной информации.

l[ist] - выводит на экран номер текущей функции и номер оператора.

l[ist] a[rgument] - выводит на экран аргументы, полученные текущей функцией.

l[ist] b[reakpoints] - выводит на экран все точки останова

l[ist] g[lobals] - выводит на экран все глобальные переменные

l[ist] l[ocals] - выводит на экран все локальные переменные для текущей функции.

l[ist] m[ap] -выводит на экран карту памяти отлаживаемой программы.

l[ist] t[raceback] -выводит на экран трассировку вызова текущей функции.

Чтобы остановить выдачу "list globals" или "list locals", пользователь может ввести любой символ (за исключением возврата каретки). Чтобы остановить распечатку большого массива и перескочить к следующей переменной, нажмите возврат каретки.

**72. Команда завершения.**

Чтобы завершить сеанс отладки и вернуться в CP/M, используется команда "quit". Эта команда не может быть сокращена.

**73. Алфавитный список команд отладки.**

Номер оператора определяется так:

sn:= <номер строки>.[<номер оператора>]

выражение определяется так:

выражение:= \* выражение первичное

первичное := целое

идентификатор

(выражение)

первичное[выражение]

первичное.идентификатор

первичное ->идентификатор

b[reak] [<имя функции>] [<номер оператора> [<счетчик>]]

Назначить точку останова. По умолчанию:

Имя функции - текущее

Номер оператора - 0

счетчик = 1

clear

Удалить все точки останова.

d[ump] выражение [количество] [формат]

Вывести элементы согласно формату. По умолчанию:

количество 1

формат i или c

синонимы: p[rint] и ","(запятая)

допустимые форматы:

c символ

р указатель

i или w целое число /слово

s строка

g[o]

Начать выполнение.

l[ist] - вывести на экран номер текущей функции и номер оператора.

l[ist] a[rgument]

Вывести на экран аргументы, полученные текущей функцией

l[ist] b[reakpoints]

вывести на экран все точки останова l[ist] g[lobals]

-вывести на экран все глобальные переменные

l[ist] l[ocals] -вывести на экран все локальные переменные для текущей функции

l[ist] m[ap] -вывести на экран карту памяти отлаживаемой

программы

l[ist] t[raceback] -вывести на экран трассировку вызова

текущей функции

quit

Возврат в cp/m

r[eset] [имя функции] [номер оператора]

Удалить точку останова

по умолчанию:

имя функции текущая функция

номер оператора 0

run

начинает выполнение, удаляя отладчик s[et] выражение значение [c]

Запись данных в память, обычно записывается 16-битовое значение; однако, если: выражение является символическим выражением, которое относится к переменной char, или значение заключено в одинарные кавычки, как '#’, или указана опция "c", тогда заносится только 8битовое значение.

t[race] [число операторов]

Трассирует выполнение, отмечая на экране выполненные операторы. По умолчанию: один оператор

u[ntrace][число операторов]

Выполняет операторы без трассировки. по умолчанию 1 оператор. Синоним:

w[alk]