**Компилятор SuperSoft C**

**Содержание**

Глава 1 Введение 3

Модульность 3

Промежуточный вывод: U-код 4

Конечный результат: код на ассемблере 4

Время выполнения в сравнении с временем компиляции 5

Рекомендуемая литература 5

Организация данного руководства 6

Глава 2 Использование компилятора SuperSoft C 7

Обсуждаемые темы 7

Поставляемые файлы 8

для операционной системой CP/M-80 8

для операционной системой CP/M-86 9

для операционных систем MS-DOS И PC-DOS 10

для операционной системы ZMOS 11

Файлы, необходимый для компиляции ваших программ 11

Форматы и последовательности команд 12

Компиляция с использованием ASM, стандартного ассемблера CP/M-80 13

Компиляция с помощью MAC, макроассемблера Digital Research 15

Компиляция с использованием ASM86, стандартного ассемблера CP/M-86 16

Компиляция с использованием MASM, ассемблера MS-DOS/PC-DOS 18

Компиляция с использованием M80, перемещающего ассемблера 19

Компиляция с использованием RMAC, перемещающего ассемблера 20

Компиляция с использованием AS, перемещающего ассемблера 21

Компиляция с использованием RASM 21

Параметры командной строки 22

Независимые от машины опции командной строки CC 22

Зависимые от машины опции командной строки CC 24

Параметры командной строки C2, COD2COD и C2I86 25

Директивы препроцессора компилятора 27

Директива #define 27

Директива #line 28

Директива #include 28

Директивы #IF, #IFDEF, #IFNDEF, #ELSE и #ENDIF 28

Директивы #ASM и #ENDASM 29

Директива #UNDEF 30

Использование функций list 30

Глава 3 Функции стандартной библиотеки SuperSoft C 32

Описание функций стандартной библиотеки SuperSoft C 33

Функции сгруппированы в соответствии с использованием 35

Список функций 36

Глава 4 Объединение исходного кода и изменение кода ассемблера 76

Метод 1: Перемещающие ассемблеры, модули и библиотеки 76

Метод 2: Директива #INCLUDE 78

Метод 3: Список имен файлов командной строки CC 79

Метод 4: Предварительная компиляция и вставка в заголовочный файл 79

Метод 5: Вырезать и вставить 80

Как переориентировать код, сгенерированный компилятором 82

Приложение A Различия между SuperSoft C и стандартным C 83

Приложение B Доступные в настоящее время компьютеры и ОС Конфигурации компилятора SuperSoft C 85

Приложение C Некоторые общие проблемы и решения 86

Приложение D Местонахождение поставляемых функций и внешних переменных 87

Приложение E Использование оверлеев в CP/M-80 89

Приложение F Сообщения об ошибках 91

Приложение G Функции С плавающей запятой двойной точности 99

Приложение H Функции длинных целых 104

# Глава 1 Введение

Компилятор SuperSoft C - это самокомпилирующийся, оптимизирующий компилятор, генерирующий окончательный выходной файл в исходном коде ассемблера. Он поддерживает практически полный язык C, за некоторыми исключениями, подробно описанными в [*Приложении A*](#_Приложение_A_Различия_1) этого руководства. В CP/M компилятору требуется два прохода; в CP/M-86 и MS-DOS оптимизация и генерация кода разбиты на отдельные фазы, что фактически делает компилятор трехпроходным. Компилятор SuperSoft C можно адаптировать к различным операционным системам и центральным процессорам (ЦП). См. [*Приложение B*](#_Приложение_B_Доступные_1) для получения списка доступных в настоящее время машин и конфигураций операционных систем. Благодаря присущей языку C переносимости и его конкретной реализации в нашем компиляторе, конфигурации для других операционных систем и машин могут быть легко и быстро разработаны.

Вместе с SuperSoft C поставляется множество функций, вызываемых пользователем. Они включают в себя множество совместимых с UNIX функций, что позволяет переносить исходный код между UNIX C и SuperSoft C с небольшими изменениями исходного кода. Предоставляется полный стандартный пакет ввода-вывода (STDIO). Этот пакет допускает файловый ввод-вывод, который не зависит от естественного размера записи целевой системы. Например, в "CP/M" программа SuperSoft C может открывать файл, искать любой байт в программе, а затем закрывать файл. Соответствующий байт и только соответствующий байт будет изменен этой последовательностью операций. Кроме того, SuperSoft предоставляет услуги по портированию.

Написание нашего компилятора на языке, который он реализует, не только облегчило его разработку и сопровождение, но и предоставило самые прямые средства его тестирования. Он прошел множество других обширных тестов, включая тот факт, что он используется в большинстве наших программных проектов. В результате SuperSoft C был протестирован на десятках тысяч уникальных строк исходного кода C.

В результате выполненных оптимизаций код, сгенерированный компилятором, имеет пространственную и временную эффективность, достаточную для программирования на уровне системы и приложений. Одним из показателей его эффективности является то, что по крайней мере для одной операционной системы CP/M, за исключением десятистрочной процедуры в ассемблерном коде, необходимой в качестве ссылки на операционную систему, весь интерфейс ввода-вывода компилятора написан на SuperSoft C.

Проектные решения, принятые в отношении генерации кода, позволяют компилятору SuperSoft C генерировать довольно хороший код для входа и выхода из подпрограмм. Эти решения позволяют нам иметь истинные регистровые переменные даже на машинах с небольшим количеством регистров. (В частности, SuperSoft C серии 8080 использует регистровую пару BC как истинную регистровую переменную.)

## Модульность

Тот факт, что каждый проход компилятора SuperSoft C является модульным, дает определенные важные преимущества. Разделение компилятора на отдельные автономные программы позволяет ему работать с относительно небольшим объемом доступной памяти и при этом поддерживать большую часть языка Си. Эта модульная структура также приводит к чистому интерфейсу между первым проходом, или синтаксическим анализатором (CC), и дополнительным проходом оптимизации (COD2COD), а также между проходом оптимизации и конечным проходом, или генератором кода (C2 или C2I36). Выходной файл одного прохода просто становится входным файлом следующего.

Модульность также облегчает адаптацию компилятора к другим машинам и операционным системам, поскольку первый модуль является машинно- и системно-независимым и только части генератора кода должны быть изменены.

## Промежуточный вывод: U-код

Первый проход компилятора, CC, принимает в качестве входных данных файл исходного кода SuperSoft C и анализирует его. В качестве вывода он генерирует файл в промежуточном коде, известном как универсальный код или U-код (подразумевающий код, не специфичный для какой-либо одной системы или машины). Поскольку одной из конструктивных особенностей компилятора было то, что вывод всего понятен человеку, U-код можно просматривать и изменять с помощью обычного текстового редактора.

Машинно-независимый оптимизатор U-кода для U-кода поставляется вместе с вашим компилятором. В CP/M-86 и MS-DOS процесс оптимизации происходит в отдельном проходе (COD2COD); в CP/M этот процесс происходит во время прохода генерации кода (C2). Оптимизатор принимает в качестве входных данных файл U-кода, сгенерированный CC. Входной файл подвергается сложному процессу оптимизации, включающему глобальную перестановку кода внутри функций, а также повторяющиеся локальные преобразования кода в "глазке" или окне определенного количества строк кода. Процесс генерации кода (C2 или C2I86) создает конечный выходной файл на языке ассемблера.

## Конечный результат: код на ассемблере

Выбор ассемблерного кода в качестве конечного результата компилятора дает ряд преимуществ, некоторые из которых имеют особое значение для программиста системного уровня. Поскольку генерируемый код намеренно не специфичен ни для одного из ассемблеров, используемых на данной машине, можно использовать практически любой аппаратно-совместимый ассемблер (включая абсолютные, а также перемещающиеся ассемблеры). Таким образом, выходные данные этого компилятора могут быть интегрированы в любую программную систему, уже написанную в исходном коде ассемблера или совместимую с ним. Программист также может вставлять строки ассемблерного кода непосредственно в исходный файл C, заключая их в скобки с директивами компилятора #ASM и #ENDASM. Вставленные таким образом строки никак не оптимизируются и не изменяются компилятором.

Использование исходного кода на ассемблере также удовлетворяет нашей проектной спецификации, требующей, чтобы результат каждого прохода был понятен человеку. Полученная в результате читаемость вывода компилятора облегчает отладку и позволяет любому пользователю видеть тип кода, созданного компилятором. Таким образом, программисту не нужно считать само собой разумеющимся, что сгенерированный код является желаемым, и он может либо изменить исходный код, либо "отполировать вручную" сгенерированный код, чтобы сделать результат более подходящим для особенно требовательного приложения.

## Время выполнения в сравнении с временем компиляции

Главный компромисс при разработке любого компилятора - это время, необходимое для компиляции программы, по сравнению со временем, необходимым для ее выполнения. Поскольку одной из основных целей разработки было создание кода, достаточно эффективного для программирования на системном уровне, мы сделали упор на скорость выполнения в ущерб скорости компиляции. (Для некоторых оптимизаций, выполняемых оптимизатором кода, требуется время, пропорциональное квадрату размера (количеству инструкций U-кода) самой большой функции C, подлежащей оптимизации. Оптимизацию можно отключить для более быстрой компиляции. Этот акцент, увеличивая время оборота во время разработки программы, делает наш компилятор полезным для гораздо более широкого круга задач программирования. Компилятор SuperSoft C уникален тем, что позволяет вам выполнять эффективное программирование на системном уровне со структурой и ясностью на относительно небольшой аппаратной системе.

## Рекомендуемая литература

Стандартная ссылка на язык:

**Brian W. Kernighan, and Dennis M. Ritchie, The C Programming Language (Englewood Cliffs, NJ: Prentice-Hall Inc.)/ 1978.**

Руководство по программированию, написанное Деннисом Ричи, главным разработчиком C, и Брайаном Керниганом, представляет собой хорошо продуманное и удобочитаемое введение в C и включает в себя Справочное руководство по C в качестве приложения. Это незаменимо для любого потенциального программиста на C. Статья, которая затрагивает эволюцию и философию языка C:

**D. M. Ritchie, et al., "The C Programming Language, "The Bell System Technical Journal, 57(6) (July-August 1978), 1991-2019.**

Учебник по языку C:

**Thomas Plum, Learning to Program in C (Cardiff, NJ: Plum Hall, 1983).**

Также полезны:

**Alan R. Seuer, The C Puzzle Book (Englewood Cliffs, NJ: Prentice-Hall Inc., 1982).**

**Jean Yates and Rabecca Thomas, A User Guide to the UNIX System (Berkeley, CA: OSBORNE/McGraw-Hill, 1982).**

**Ann and Nico Lomuto, A Unix Primer (Englewood Cliffs, NJ: Prentice-Hall Inc., 1983).**

## Организация данного руководства

В следующих главах представлена информация, которая вам понадобится для использования компилятора SuperSoft C. В [*главе 2*](#_Глава_2_Использование) представлены инструкции по вызову директив компилятора и препроцессора. Описание функций стандартной библиотеки, поставляемых с компилятором, приведено в [*главе 3*](#_Глава_3_Функции). В [*главе 4*](#_Глава_4_Объединение) описывается, как вставить код в библиотеку времени выполнения генератора кода. Различия между SuperSoft C и стандартной версией 7 UNIX C перечислены в [*Приложении A*](#_Приложение_A_Различия). Доступные в настоящее время конфигурации компьютеров и операционных систем компилятора SuperSoft C перечислены в [*Приложении B*](#_Приложение_B_Доступные). В [*Приложении C*](#_Приложение_C_Некоторые) описаны некоторые общие проблемы и их решения. [*Приложение D*](#_Приложение_D_Местонахождение) состоит из списка поставляемых функций, организованного в соответствии с файлом, в котором они находятся. В [*Приложении E*](#_Приложение_E_Использование) обсуждается использование оверлейных программ. Сообщения об ошибках программы перечислены в [*Приложении F*](#_Приложение_F_Сообщения). [*Приложение G*](#_Приложение_G_Функции) описывает функции над числами с плавающей запятой двойной точности. [*Приложение H*](#_Приложение_H_Функции), функции длинных целых чисел.

Мы надеемся, что компилятор SuperSoft C станет одним из ваших самых полезных инструментов программирования. Мы приветствуем любые комментарии, которые у вас могут возникнуть по поводу компилятора или его документации.

# Глава 2 Использование компилятора SuperSoft C

## Обсуждаемые темы

В этой главе представлена информация, необходимая для наилучшего использования компилятора SuperSoft C. Обсуждаются следующие темы: файлы, поставляемые на диске компилятора, какие из этих файлов вам понадобятся для компиляции ваших программ, другое программное обеспечение, которое вам понадобится, синтаксис командных строк компилятора, использование нашего компилятора в конкретной операционной системе, параметры командной строки, доступные для каждого прохода, и директивы препроцессора компилятора, поддерживаемые в SuperSoft C.

**Для вашей собственной защиты** сделайте копию вашего диска компилятора SuperSoft C, когда вы впервые получите его. Храните оригинал в надежном месте в качестве мастер-диска и для всех последующих работ используйте только копию.

## Поставляемые файлы

### для операционной системой CP/M-80

Если у вас есть версия компилятора CP/M-80, вы должны найти следующие файлы в своей рабочей копии:

|  |  |  |
| --- | --- | --- |
| **CC.COM** | : | первый проход компилятора: лексический анализатор |
| **C2.COM** | : | второй проход: оптимизатор/генератор кода |
| **C2.RH** | : | заголовочный файл библиотеки времени выполнения, используемый для ASM |
| **C2.RT** | : | файл трейлера времени выполнения, используемый для ASM |
| **C2PRE.ASM** | : | источник заголовка времени выполнения для L80 |
| **C2POST.ASM** | : | исходный код трейлера времени выполнения для L80 |
| **C2PRE.REL** | : | заголовок времени выполнения для L80 |
| **C2POST.REL** | : | трейлер времени выполнения для L80 |
| **\*.ASM** | : | различные другие источники времени выполнения |
| **MDEP.C** | : | #ASM/#ENDASM версия среды выполнения |
| **C.SUB** | : | файл CP/M SUBMIT для компиляции |
| **CBRACK.H** | : | определения верхнего регистра для клавиатур без нижнего регистра |
| **CUSTOMIZ.H** | : | параметры времени компиляции библиотеки C |
| **STDIO.H** | : | заголовок функций стандартного ввода/вывода |
| **STDIO.C** | : | функции стандартного ввода/вывода |
| **ALLOC.C** | : | функции распределения динамической памяти |
| **CRUNT2.C** | : | части среды выполнения на языке C |
| **FUNC.C** | : | вспомогательные функции |
| **FORMATION.C** | : | printf, scanf и др. функции |
| **LONG.C** | : | длинные целочисленные функции |
| **DOUBLE.C** | : | функции с плавающей запятой удвоенной точности |
| **BCD80.C** | : | поддержка языка ассемблера для DOUBLE.C |
| **SAMP? .C** | : | примеры программ, которые тестируют возможности C |
| **LIBC.REL** | : | библиотека C в перемещаемом формате |
| **\*.SUB** | : | различные SUBMIT файлы |
| **SH.COM** | : | SUBMIT с расширенными возможностями |

CC.COM и C2.COM являются исполняемыми файлами компилятора в CP/M-80

### для операционной системой CP/M-86

Если у вас есть версия компилятора CP/M-86, вы должны найти следующие файлы в своей рабочей копии:

|  |  |  |
| --- | --- | --- |
| **CC.CMD** | : | первый проход компилятора: лексический анализатор |
| **COD2COD.CMD** | : | второй проход: оптимизатор |
| **C2I86.CMD** | : | третий проход: генератор кода |
| **C2I86.RH** | : | заголовочный файл библиотеки времени выполнения, используемый для ASM86 |
| **C2I86.RT** | : | файл трейлера времени выполнения, используемый для ASM86 |
| **C2PRE.ASM** | : | источник заголовка времени выполнения |
| **C2POST.ASM** | : | исходный код трейлера времени выполнения |
| **\*.ASM** | : | различные другие источники времени выполнения |
| **MDEP.C** | : | #ASM/#ENDASM версия среды выполнения |
| **C.SUB** | : | файл CP/M SUBMIT для компиляции |
| **CBRACK.H** | : | определения верхнего регистра для клавиатур без нижнего регистра |
| **CUSTOMIZ.H** | : | параметры времени компиляции библиотеки C |
| **STDIO.H** | : | заголовок функций стандартного ввода/вывода |
| **STDIO.C** | : | функции стандартного ввода/вывода |
| **ALLOC.C** | : | функции распределения динамической памяти |
| **CRUNT2.C** | : | части среды выполнения на языке C |
| **FUNC.C** | : | вспомогательные функции |
| **FORMATION.C** | : | printf, scanf и др. функции |
| **LONG.C** | : | длинные целочисленные функции |
| **DOUBLE.C** | : | функции с плавающей запятой удвоенной точности |
| **BCD80.C** | : | поддержка языка ассемблера для DOUBLE.C |
| **SAMP?.C** | : | примеры программ, которые тестируют возможности C |

CC.CMD, COD2COD.CMD и C2I86.CMD - это исполняемые файлы компилятора в CP/M-86. Второй проход разделен на COD2COD (оптимизатор) и C2I86 (генератор кода), который мы называем третьим проходом.

### для операционных систем MS-DOS И PC-DOS

Если у вас есть версия компилятора для MS-DOS или PC-DOS, вы найдете эти файлы в своей рабочей копии:

|  |  |  |
| --- | --- | --- |
| **CC.EXE** | : | первый проход компилятора: лексический анализатор |
| **COD2COD.EXE** | : | второй проход: оптимизатор |
| **C2I86.EXE** | : | третий проход: генератор кода |
| **C2I86.RTB** | : | заголовочный файл библиотеки времени выполнения, используемый для MASM |
| **C2I86.RTM** | : | файл трейлера времени выполнения, используемый для MASM |
| **C2PRE.ASM** | : | источник заголовка времени выполнения для LINK |
| **C2POST.ASM** | : | исходный код трейлера времени выполнения для LINK |
| **MDEP.C** | : | #ASM/#ENDASM версия среды выполнения |
| **C.BAT** | : | командный файл MS-DOS для компиляции |
| **CBRACK.H** | : | определения верхнего регистра для клавиатур без нижнего регистра |
| **CUSTOMIZ.H** | : | параметры времени компиляции библиотеки C |
| **STDIO.H** | : | заголовок функций стандартного ввода/вывода |
| **STDIO.C** | : | функции стандартного ввода/вывода |
| **ALLOC.C** | : | функции распределения динамической памяти |
| **CRUNT2.C** | : | части среды выполнения на языке C |
| **FUNC.C** | : | вспомогательные функции |
| **FORMATION.C** | : | printf, scanf и др. функции |
| **LONG.C** | : | длинные целочисленные функции |
| **DOUBLE.C** | : | функции с плавающей запятой удвоенной точности |
| **BCD80.C** | : | поддержка языка ассемблера для DOUBLE.C |
| **SAMP? .C** | : | примеры программ, которые тестируют возможности C |
| **LIBC.BAT** |  | Пакетный файл для компиляции библиотечных функций |
| **LIBC.LIB** |  | Библиотека C в перемещаемом формате |

CC.EXE, COD2COD.EXE и C2I36.EXE - это исполняемые файлы компилятора под MS-DOS и PC-DOS. Второй проход разделен на COD2COD (оптимизатор) и C2I86 (генератор кода), который мы называем третьим проходом.

Для версий программы CP/M, CP/M-86 и MS-DOS следующие: C2PRE, C2POST, STDIO.H, STDIO.C, FORMATIO.C, ALLOC.C, FUNC.C, CRUNT2.C, LONG.C и DOUBLE.C - файлы встроенных функций. (Остальные файлы на диске с расширением \*.C - это программы в исходном коде SuperSoft C, представленные в качестве примеров.)

### для операционной системы ZMOS

Если у вас ZMOS-версия компилятора, вы должны найти в своей рабочей копии следующие файлы:

|  |  |  |
| --- | --- | --- |
| **CC.IMAG** | : | первый проход компилятора: лексический анализатор |
| **C2Z8001.IMAG** | : | второй проход: оптимизатор/ генератор кода |
| **WHEADER.REG** | : | файла заголовка библиотеки времени выполнения вывода на C2Z8001 |
| **WTRA1LER.REG** | : | файла трейлера времени выполнения вывода на C2Z8001 |
| **ZSTDIO.REG** | : | стандартные функции ввода/вывода |
| **ALLOC.REG** | : | функции распределения динамической памяти |
| **CRUNT2.REG** | : | части среды выполнения на языке C |
| **FUNC.REG** | : | вспомогательные функции |
| **FORMATIO.REG** | : | printf, scanf и др. |
| **LONG.REG** | : | длинные целые функции |
| **DOUBLE.REG** | : | функции с плавающей запятой удвоенной точности |
| **SAMP?.REG** | : | примеры программ, которые тестируют возможности C |

CC.IMAG и C2Z8001.IMAG - это исполняемые файлы компилятора под ZMOS. Файлы с расширением ".REG" содержат файлы заголовков, библиотечные функции или, в случае SAMP?.REG, примеры программ.

## Файлы, необходимый для компиляции ваших программ

Для компиляции программы вам потребуются исполняемые файлы, соответствующие вашей конкретной операционной системы. Кроме того, вам потребуются соответствующие файлы заголовков и трейлеров, а также файлы, содержащие любые желаемые библиотечные функции. Этап генерации кода компилятора автоматически включает все сегменты кода и функции, содержащиеся в его библиотеке времени выполнения, а также файлы заголовков и трейлеров в вашу программу во время компиляции. Если вы используете перемещаемый ассемблер, вам понадобится соответствующий файл трейлера (C2.RTM для CP/M-80) и LIBC. Используя перемещаемый ассемблер, вам нужно будет связать функции библиотеки времени выполнения. Файл с расширением ".H" или ".C" потребуется только в том случае, если ваша программа вызывает одну или несколько своих функций. Так как функции, содержащиеся в этих файлах, являются такими базовыми строительными блоками, большинство программ будет вызывать их значительное количество. Поэтому, если вашей программе требуется какая-либо из функций, определенных в этих файлах ".C" или ".H", вы должны убедитесь, что он включает в себя все необходимые определения функций и данных из этих файлов и, в свою очередь, включает ту же информацию из функций, которые эти функции вызывают. Пять методов достижения этого описаны в разделе *главы 4* "Включение стандартных библиотечных функций". Также см. [*Приложение D*](#_Приложение_D_Местонахождение_1), где приведены сведения о том, какие поставляемые функции C находятся в каких файлах.

Поскольку окончательный вывод этого компилятора находится в исходном коде ассемблера вашего компьютера, вам также понадобится ассемблер, совместимый с вашей операционной системой и оборудованием, а также программное обеспечение, необходимое для загрузки и запуска программ в вашей системе.

## Форматы и последовательности команд

Чтобы скомпилировать исходную программу SuperSoft C, каждый проход должен запускаться в следующей последовательности:

**passname filename.ext ... options**

Имя файла (**filename**) относится к префиксной части спецификации файла, требуемой вашей операционной системой. Имя прохода (**passname**) относится к имени прохода компилятора C.

Параметры командной строки для каждого прохода следует разделять пробелами. Никаких опций указывать не нужно, поскольку условия по умолчанию определяют обычно желаемый режим работы для компилятора. Однако, если вы запускаете компилятор на ЦП, отличном от 8080, 8085, 8086 или Z80 (серия 8080), или ориентируетесь на другую машину, а не на хост-машину, вам, возможно, придется установить зависящие от машины параметры , +J, -I и -P, описанные на странице 31, в их значения, соответствующие вашему процессору.

При вызове CC вы можете указать в командной строке любое количество имен файлов, разделенных пробелами. Указанные файлы не обязательно должны содержать полные функции, поскольку все они будут проанализированы в указанном порядке, как если бы они были одним файлом. Выходному файлу с U-кодом будет присвоено первое имя файла. Вы можете указать только одно имя файла при выполнении других проходов. Каждый проход генерирует файл с тем же именем, которое вы указали, но с другим расширением.

По соглашению файлы исходного кода C имеют расширение или суффикс. ".C", хотя любое расширение является допустимым. CC автоматически предоставляет расширение ".COD" для своих выходных файлов U-кода. Расширение ".U" используется для оптимизации U-кода. Расширение ".ASM" используется для вывода на ассемблере. Другие проходы имеют аналогичные характеристики. Эти:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Операционная система** | **Имя прохода** | **Входной файл** |  | **Выходной файл** |
| CP/M-80, CP/M-86 и MS-DOS | CC | f.C | 🡪 | f.COD |
| CP/M-80 | C2 | f.COD | 🡪 | f.ASM |
| CP/M-86 и MS-DOS | COD2COD | f.COD | 🡪 | f.U |
| ZMOS | C2Z8001 | f.COD | 🡪 | CASM.REG |
| UNIX | C2Z8002 | f.COD | 🡪 | f.ASM |
| UNIX | C2Z8002 | f.U | 🡪 | f.ASM |
| CP/M-86 | C2I86 | f.COD | 🡪 | f.A86 |
| CP/M-86 | C2I86 | f.U | 🡪 | f.A86 |
| MS-DOS | C2I86 | f.COD | 🡪 | f.ASM |
| MS-DOS | C2I86 | f.U | 🡪 | f.ASM |

После того, как компилятор SuperSoft C выполнит свою работу, он создает файл, который следует запустить с помощью ассемблера. Выбор ассемблеров широк, в зависимости от операционной системы и ваших предпочтений.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Операционная система** | **Имя прохода** | **Входной файл** |  | **Выходной файл** |
| CP/M-80 | ASM | f.C | 🡪 | f.HEX |
| CP/M-80 | MAC | f.COD | 🡪 | f.HEX |
| CP/M-80 | M80 | f.ASM | 🡪 | f.REL |
| CP/M-80 | RMAC | f.ASM | 🡪 | f.REL |
| CP/M-86 | ASM86 | f.A86 | 🡪 | f.H86 |
| MS-DOS | MASM | f.ASM | 🡪 | f.OBL |
| UNIX | AS | f.ASM | 🡪 | f.O |
| ZMOS | RASM | f.REG | 🡪 | f.IMAG |

После прохождения компиляции выходной файл должен быть преобразован в исполняемый файл. На этом процесс компиляции завершается. Вот несколько примеров такого завершения, организованных операционной системой и преобразованием файлов.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Операционная система** | **Имя прохода** | **Входной файл** |  | **Выходной файл** |
| CP/M-80 | LOAD | f.HEX | 🡪 | f.COM |
| CP/M-80 | L80 | f.HEX | 🡪 | f.COM |
| CP/M-80 | ELINK | f.REL | 🡪 | f.COM |
| CP/M-80 | LINK80 | f.REL | 🡪 | f.COM |
| CP/M-86 | GENCMD | f.H86 | 🡪 | f.CMD |
| MS-DOS | LINK | f.OBJ | 🡪 | f.EXE |
| UNIX | LD | f.O | 🡪 | f |

В качестве примера процедуры, которой необходимо следовать, чтобы скомпилировать и запустить программу SuperSoft C в конкретной машине и операционной системе, мы опишем процедуры, необходимые для различных операционных систем.

Мы предположим, что вы чувствуете себя готовым скомпилировать и запустить свою первую программу на языке Си, SAMP1.C. после первой проверки того, что все необходимые файлы доступны на вашем диске и что ваша программа включает в себя все необходимые данные и определения функций, вы готовы начать.

### Компиляция с использованием ASM, стандартного ассемблера CP/M-80

Чтобы преобразовать SAMP1.C в исполняемый командный файл, чтобы его можно было запускать под CP/M-80, вы должны последовательно ввести каждую из перечисленных ниже команд. Эта процедура аналогична процедуре для MAC, описанной ниже. Убедитесь, что файлы C2.RH и C2.RT находятся на диске по умолчанию (обычно на том же диске, что и C2.COM). Выполнение каждой из этих команд CP/M начинается, как только вы нажимаете RETURN в конце строки. Введите:

**CC SAMP1.C**

**C2 SAMP1.COD**

**ASM SAMP1**

**LOAD SAMP1**

**SAMP1**

Каждая из приведенных выше команд вызывает создание нового файла на диске, содержащем SAMP1.C. Имена файлов соответственно:

SAMP1.COD ; вывод из CC

SAMP1.ASM ; вывод из C2

SAMP1.HEX ; вывод из ASM

SAMP1.COM ; вывод из LOAD

Последний созданный файл - это исполняемый командный файл для вашей программы. Это единственная форма, в которой ваша программа может работать под CP/M. (Остальные могут быть удалены.) После создания этого файла вам нужно только ввести:

**SAMP1**

Затем нажмите RETURN, и выполнение вашей программы начнется.

CC, C2 и ASM также могут генерировать сообщения об ошибках, указывающие на дефекты в вашей программе. Хотя эти сообщения в значительной степени говорят сами за себя, их полное объяснение можно найти в [*Приложении F*](#_Приложение_F_Сообщения_1) данного руководства. Документация CP/M предоставляет информацию о сообщениях об ошибках, генерируемых ASM. Два типа сообщений от ASM могут иметь корни в плохо сформированных программах на C: (1) фазовые ошибки и многократно определенные метки обычно вызываются переопределением внешнего C (включая функции C). Обратите внимание, что внешние элементы имеют ограниченное количество значимых ведущих символов. Это число зависит от используемого ассемблера (и загрузчика). (2) Неопределенные метки обычно указывают на неопределенные внешние элементы C (включая функции C).

В процессе разработки программ вы, вероятно, будете повторять только что описанную процедуру (без изменений, кроме указанного имени файла) огромное количество раз. Если бы вы могли выполнить всю эту процедуру с помощью одной команды, это значительно сэкономило бы время. В CP/M команда SUBMIT и файлы SUBMIT предоставляют средства для достижения этого.

Соответствующий файл SUBMIT с именем C.SUB должен содержать следующие командные строки

**CC $1.C**

**C2 $1.COD +ASM**

**ASM $1**

**LOAD $1**

**$1**

где "$1" - символический параметр, который позже будет заменен первым фактическим параметром команды SUBMIT.

После того, как вы создали такой файл на своем диске, введите:

**SUBMIT C SAMP1**

Затем нажмите RETURN. Если CP/M находит C.SUB на текущем диске, будут выполнены следующие пять команд (SAMP1 заменяется на каждое вхождение "$1"):

**CC SAMP1.C**

**C2 SAMP1.COD**

**ASM SAMP1**

**LOAD SAMP1**

**SAMP1**

Для получения дополнительной информации о файлах SUBMIT обратитесь к документации CP/M или к одному из нескольких доступных руководств CP/M.

### Компиляция с помощью MAC, макроассемблера Digital Research

Чтобы преобразовать SAMP1.C в исполняемый командный файл с помощью Digital Research ассемблер MAC и фактически запустить этот файл под CP/M-80, вы должны последовательно ввести каждую из перечисленных ниже команд. Эта процедура аналогична процедуре для ASM. Убедитесь, что файлы C2.RH и C2.RT находятся на диске по умолчанию (обычно на том же диске, что и C2.COM). Выполнение каждой из этих команд CP/M начинается, как только вы нажимаете RETURN в конце строки. Введите:

**CC SAMP1.C**

**C2 SAMP1.COD**

**MAC SAMP1**

**LOAD SAMP1**

**SAMP1**

Каждая из вышеперечисленных команд вызывает на диске файл SAMP1.C. Имена этих файлов соответственно:

SAMP1.COD ; вывод из CC

SAMP1.ASM ; вывод из C2

SAMP1.HEX ; вывод из MAC

SAMP1.COM ; вывод из LOAD

Последний созданный файл - это исполняемый командный файл для вашей программы. Это единственная форма, в которой ваша программа: может быть запущена под CP / M. (Остальные могут быть удалены.) После создания этого файла вам нужно только ввести:

**SAMP1**

Затем нажмите RETURN, и выполнение вашей программы начнется.

CC, C2 и ASM также могут генерировать сообщения об ошибках, указывающие на дефекты в вашей программе. Хотя эти сообщения в значительной степени говорят сами за себя, их полное объяснение можно найти в [*Приложении F*](#_Приложение_F_Сообщения_2) данного руководства. Документация CP/M предоставляет информацию о сообщениях об ошибках, генерируемых ASM. Два типа сообщений от ASM могут иметь корни в плохо сформированных программах на C: (1) фазовые ошибки и многократно определенные метки обычно вызываются переопределением внешнего C (включая функции C). Обратите внимание, что внешние элементы имеют ограниченное количество значимых ведущих символов. Это число зависит от используемого ассемблера (и загрузчика). (2) Неопределенные метки обычно указывают на неопределенные внешние элементы C (включая функции C).

В процессе разработки программ вы, вероятно, будете повторять только что описанную процедуру (без изменений, кроме указанного имени файла) огромное количество раз. Если бы вы могли выполнить всю эту процедуру с помощью одной команды, это значительно сэкономило бы время. В CP/M команда SUBMIT и файлы SUBMIT предоставляют средства для достижения этого.

Соответствующий **файл SUBMIT** с именем C.SUB должен содержать следующие командные строки

**CC $1.C**

**C2 $1.COD +ASM**

**MAC $1**

**LOAD $1**

**$1**

где "$1" - символический параметр, который позже будет заменен первым фактическим параметром команды SUBMIT.

После того, как вы создали такой файл на своем диске, введите:

**SUBMIT C SAMP1**

Затем нажмите RETURN. Если CP/M находит C.SUB на текущем диске, будут выполнены следующие пять команд (SAMP1 заменяется на каждое вхождение "$1"):

**CC SAMP1.C**

**C2 SAMP1.COD**

**MAC SAMP1**

**LOAD SAMP1**

**SAMP1**

Для получения дополнительной информации о файлах SUBMIT обратитесь к документации CP/M или к одному из нескольких доступных руководств CP/M.

### Компиляция с использованием ASM86, стандартного ассемблера CP/M-86

Чтобы преобразовать SAMP1.C в исполняемый командный файл, чтобы его можно было запустить под управлением CP/M-86, вы должны последовательно ввести каждую из перечисленных ниже команд. Введите:

**CC SAMP1.C**

**COD2COD SAMP1.COD**

**C2I86 SAMP1.U**

**ASM86 SAMP1**

**GENCMD SAMP1 DATA[X1000]**

Каждая из вышеперечисленных команд вызывает создание нового файла на диске, содержащем SAMP1.C. Имена этих файлов соответственно:

SAMP1.COD ; вывод из CC

SAMP1.U ; вывод из COD2COD

SAMP1.A86 ; вывод из C2I86

SAMP1.H86 ; вывод из ASM86

SAMP1.CMD ; вывод из GENCMD

Последний созданный файл-это исполняемый командный файл для вашей программы. Это единственная форма, в которой ваша программа может быть запущена под CP/M-86. (Остальные могут быть удалены.) Как только этот файл будет создан, вам нужно только ввести:

**SAMP1**

Затем нажмите RETURN, и выполнение вашей программы начнется.

CC, COD2COD, C2I86 и ASMS6 также могут генерировать сообщения об ошибках, указывающие на дефекты в вашей программе. Хотя эти сообщения в основном не требуют пояснений, их полное объяснение можно найти в [*Приложении F*](#_Приложение_F_Сообщения_3) к настоящему руководству. Кроме того, документация CP/M-86 предоставляет информацию о сообщениях об ошибках, генерируемых ASM86. Два типа сообщений от ASM86 могут иметь корни в плохо сформированных программах на C: (1) фазовые ошибки и многократно определенные метки обычно вызваны переопределением внешнего C (включая функции C). Обратите внимание, что внешние элементы имеют ограниченное количество значимых ведущих символов. Это число зависит от используемого ассемблера (и загрузчика). (2) Неопределенные метки обычно указывают на неопределенные внешние элементы C (включая функции C). В процессе разработки программ вы, вероятно, будете повторять только что описанную процедуру - без изменений, за исключением указанного имени файла - огромное количество раз. Было бы сэкономить время, если бы вы могли выполнить всю эту процедуру с помощью одной команды. В CP/M-86 команда SUBMIT и файлы SUBMIT предоставляют средства для этого.

Соответствующий файл SUBMIT с именем C.SUB будет содержать следующие командные строки

**CC $1.C**

**COD2COD $1.COD**

**C2I86 $11.U**

**ASM86 $1**

**GENCMD $1 DATA[X1000]**

где "$1" - символический параметр, который позже будет заменен первым фактическим параметром команды SUBMIT.

После того, как вы создали такой файл на своем диске, введите:

**SUBMIT C SAMP1**

Затем нажмите RETURN. Если CP/M находит C.SUB на текущем диске, будут выполнены следующие пять команд (SAMP1 заменяется на каждое вхождение "$1"):

**CC SAMP1.C**

**COD2COD SAMP1.COD**

**C2I86 SAMP1.U**

**ASM86 SAMP1**

**GENCMD SAMP1 DATA[X1000]**

Для получения дополнительной информации о файлах SUBMIT обратитесь к документации CP/M-86 или к одному из нескольких имеющихся руководств по CP/M-86.

### Компиляция с использованием MASM, ассемблера MS-DOS/PC-DOS

Чтобы преобразовать SAMP1.C в исполняемый командный файл, чтобы его можно было запустить в MS-DOS, вы должны последовательно ввести каждую из перечисленных ниже команд. Введите:

**CC SAMP1.C ;**

**COD2COD SAMP1.COD**

**C2I86 SAMP1.U -ASM +MSDOS**

**MASM SAMP1.ASM**

**LINK C2PRE+SAMP1+C2POST,SAMP1,NUL,LIBC**

**SAMP1**

Каждая из вышеперечисленных команд вызывает создание нового файла на диске, содержащем SAMP1.C. Имена этих файлов соответственно:

SAMP1.COD ; вывод из CC

SAMP1.U ; вывод из COD2COD

SAMP1.ASM ; вывод из C2I86

SAMP1.OBJ ; вывод из MASM

SAMP1.EXE ; вывод из LINK

Последний созданный файл-это исполняемый командный файл для вашей программы. Это единственная форма, в которой ваша программа может быть запущена под MS-DOS. (Остальные могут быть удалены.) Как только этот файл будет создан, вам нужно только ввести:

**SAMP1**

Затем нажмите RETURN, и выполнение вашей программы начнется.

CC, COD2COD, C2I86 и MASM также могут генерировать сообщения об ошибках, указывающие на дефекты в вашей программе. Хотя эти сообщения в значительной степени говорят сами за себя, полное их объяснение можно найти в [*Приложении F*](#_Приложение_F_Сообщения_4) к настоящему руководству. Кроме того, документация MS-DOS предоставляет информацию о сообщениях об ошибках, генерируемых MASM. Два типа -сообщений от В MS-DOS пакетная команда и пакетные файлы предоставляют средства для достижения этой цели. могут иметь корни в плохо сформированных программах на C: (1) фазовые ошибки и многократно определенные метки обычно вызваны переопределением внешнего C (включая функции C). Обратите внимание, что внешние элементы имеют ограниченное количество значимых ведущих символов. Это число зависит от используемого ассемблера (и загрузчика). (2) Неопределенные метки обычно указывают на неопределенные внешние элементы C (включая функции C).

В ходе разработки программ вы, вероятно, будете повторять только что описанную процедуру - без изменений, за исключением указанного имени файла - огромное количество раз. если бы вы могли выполнить всю эту процедуру с помощью одной команды, это значительно сэкономило бы время. В MS-DOS пакетная команда и пакетные файлы предоставляют средства для достижения этой цели.

Соответствующий пакетный файл с именем C.BAT будет содержать следующие командные строки

**CC %1.C ;**

**COD2COD %1.COD**

**C2I86 %1.U -ASM +MSDOS**

**MASM %1.ASM**

**LINK C2PRE+%1+C2POST,%1,NUL,LIBC**

**%1**

где "%1" - это символический параметр, который позже будет заменен первым фактическим параметром пакетной команды.

После того, как вы создали такой файл на своем диске, введите

**C SAMP1**

Затем нажмите RETURN.

Для получения дополнительной информации о пакетных файлах обратитесь к документации MS-DOS или одному из нескольких доступных руководств MS-DOS.

### Компиляция с использованием M80, перемещающего ассемблера

Чтобы преобразовать SAMP1.C в исполняемый командный файл, чтобы его можно было использовать в CP/M-80, вы должны последовательно ввести каждую из перечисленных ниже команд. Эта процедура аналогична описанной выше, за исключением того, что создаются перемещаемые модули и модули соединяются вместе.

Введите:

**CC SAMP1.C**

**C2 SAMP1.COD**

**M80 =SAMP1.ASM**

**L80 C2PRE,SAMP1,LIBC/S,C2POST,SAMP1/N/E/Y**

**SAMP1**

C2POST должен быть указан так, как показано (последний модуль).

В качестве альтернативы строку с L80 можно заменить с помощью ELINK от SuperSoft:

**ELINK BC(SAMP1);C2PRE,SAMP1;SR(LIBC);IN(C2POST);EN;**

Каждая из вышеперечисленных команд вызывает создание новых файлов на текущем диске. Это:

SAMP1.COD ; вывод из CC

SAMP1.ASM ; вывод из C2

SAMP1.REL ; вывод из M80

SAMP1.COM ; вывод из L80 или ELINK

Использование описанной выше процедуры потребует наличия следующих файлов в перемещаемом формате:

C2PRE.REL ; заголовок времени выполнения из C2PRE.ASM

C2POST.REL ; трейлер времени выполнения из C2POST.ASM

LIBC.REL ; библиотека C

LIBC.REL - это библиотека, созданная путем перекомпиляции следующих источников: ALLOC.C, STDIO.C, CRUNT2.C, FUNC.C и FORMATIO.C. Вам также понадобится файл C2.RTM на диске.

Соответствующий файл SUBMIT для перемещающих ассемблеров с использованием L80:

**CC $1.C**

**C2 $1.COD**

**M80 =$1.ASM**

**L80 C2PRE,$1,LIBC/S,C2POST,$1/N/E/Y**

**$1**

или если используется ELINK

**CC $1.C**

**C2 $1.COD**

**M80 =$1.ASM**

**ELINK BC($1);C2PRE,$1;SR(LIBC);IN(C2POST);EN;**

**$1**

### Компиляция с использованием RMAC, перемещающего ассемблера

Чтобы преобразовать SAMP1.C в исполняемый командный файл, чтобы его можно было использовать в CP/M-80, вы должны последовательно ввести каждую из перечисленных ниже команд. Эта процедура также создает модули REL и связывает их вместе, как в примере M80 и L80 выше. Ниже приводится процедура:

**CC SAMP1.C**

**C2 SAMP1.COD -OFILESAMP1.MAC**

**RMAC SAMP1**

**LINK SAMP1,C2PRE,LIBC[S],C2POST**

**SAMP1**

C2POST должен быть указан так, как показано (последний модуль).

В качестве альтернативы строку с LINK80 можно заменить с помощью ELINK от SuperSoft:

**ELINK BC(SAMP1);C2PRE,SAMP1;SR(LIBC);IN(C2POST);EN;**

C2POST должен быть указан последним. Каждая из вышеперечисленных команд вызывает создание нового файла на диске, на котором в данный момент выполнен вход. Эти файлы следующие:

SAMP1.COD ; вывод из CC

SAMP1.MAC ; вывод из C2

SAMP1.REL ; вывод из RMAC

SAMP1.COM ; вывод из LINK80 или ELINK

Использование описанной выше процедуры потребует наличия следующих файлов в перемещаемом формате RMAC REL:

C2PRE.REL ; заголовок времени выполнения из C2PRE.ASM

C2POST.REL ; трейлер времени выполнения из C2POST.ASM

LIBC.REL ; библиотека C

LIBC.REL - это библиотека, созданная путем перекомпиляции следующих источников: ALLOC.C, STDIO.C, CRUNT2.C, FUNC.C и FORMATIO.C.

### Компиляция с использованием AS, перемещающего ассемблера

Чтобы преобразовать SAMP1.C в исполняемый командный файл, чтобы его можно было использовать в UNIX, вам следует последовательно ввести каждую из перечисленных ниже команд. Эта процедура также создает перемещаемые модули и связывает их вместе, как в примере M80 и L80 выше. Ниже приводится процедура:

**CC CSAMP1.C**

**COD2COD SAMP1.COD**

**C2Z8002 SAMP1.U**

**AS SAMPI.ASM -O SAMP1.O**

**LD SAMP1.O C2PRE.O LIBC.A C2POST.O**

**SAMP1**

C2POST.O должен быть указан последним. Каждая из вышеперечисленных команд вызывает создание нового файла на диске, на котором в данный момент выполнен вход. Эти файлы следующие:

SAMP1.COD ; вывод из CC

SAMP1.U ; вывод из COD2COD

SAMP1.ASM ; вывод из C2Z8002

SAMP1.O ; вывод из AS

SAMP1 ; вывод из LD

Использование описанной выше процедуры потребует наличия следующих файлов в перемещаемом формате UNIX .O :

C2PRE.O ; заголовок времени выполнения из C2PRE.ASM

C2POST.O ; трейлер времени выполнения из C2POST.ASM

LIBC.A ; библиотека C

LIBC.A - это библиотека, созданная путем перекомпиляции следующих источников: ALLOC.C, STDIO.C, CRUNT2.C, FUNC.C и FORMATIO.C.

### Компиляция с использованием RASM

Чтобы преобразовать SAMP1.C в исполняемый командный файл, чтобы его можно было использовать в ZMOS, вы должны последовательно ввести каждую из перечисленных ниже команд. Эта процедура также создает модули IMAG и связывает их вместе, как в примере ASM. Ниже приведен порядок действий:

**CC SAMP1.C**

**C2Z8001 SAMP1.COD**

**RASM SAMP1**

**SAMP1**

Каждая из вышеперечисленных команд вызывает создание нового файла на текущем размещенном на диске. Эти файлы следующие:

SAMP1.COD ; вывод из CC

SAMP1.REG ; вывод из COD2COD

SAMP1.IMAG ; вывод из C2Z8002

Вам также потребуются файлы WHEADER.REG и WTRAILER.REG, имеющиеся на диске.

## Параметры командной строки

### Независимые от машины опции командной строки CC

|  |  |
| --- | --- |
| +A | помещает адрес последней ячейки памяти, использованной CC, перед каждой строкой исходного кода C, вставленной в качестве комментария в выходной файл. Этот параметр не действует, если также не выбран параметр +L. Эта информация о распределении памяти помогает определить, когда компилятору не хватает памяти. По умолчанию -A. |
| -BUFSIZ | за которым сразу следует десятичное целое число. Это целое число становится размером буфера для операций ввода-вывода. Чем больше это значение, тем быстрее выполняется компиляция. Меньшие размеры потребляют меньше памяти. Число 123 - это примерно минимум для большинства систем. Этот размер действует как для входных, так и для выходных буферов (см.-WBUFSIZ ниже.) Если этот параметр используется, выбранное значение будет напечатано на консоли. По умолчанию - BUFSIZ1024. |
| +CO | принудительно выводит на консоль. По умолчанию -CO (вывод в filename.COD). |
| +CR | добавляет возврат каретки к каждой сгенерированной строке вывода U-кода. Некоторые текстовые редакторы (например, CP/M ED) ожидают этот формат. По умолчанию -CR. |
| -D | за которым сразу (без пробелов) следует идентификатор, определяет этот идентификатор так же, как если бы для этого идентификатора был обнаружен #DEFINE. Полезно для директив препроцессора #IFDEF и #ENDIF. По умолчанию идентификатор не задан заранее. |
| +DDT | включает дополнительную отладочную информацию в выходной файл. По умолчанию -DDT. |
| +F | сбрасывает (записывает в файл) все строки U-кода, созданные сразу после обработки каждой строки исходного кода C. Если выбраны опции +L и +F, то за каждой строкой комментария исходного кода, помещенной в выходной файл, сразу же следуют все строки U-кода, сгенерированные из нее. Выбор этого параметра замедляет выполнение как CC, так и C2 (или C2I86) и запрещает определенные оптимизации, обычно выполняемые C2 (или COD2COD). По умолчанию -F. |
| -G | отключает генерацию U-кода. Эта опция полезна для проверки синтаксиса файла исходного кода. Выбор этой опции и опции +L ниже также приведет к созданию пронумерованного списка исходного кода C с расширенными определениями макросов. По умолчанию создается U-код. По умолчанию +G. |
| +L | перечисляет каждую строку исходного кода C как комментарий в выходном файле U-кода. Исходные строки сгруппированы по функциям и, таким образом, лишь слабо связаны с соответствующими строками U-кода. Макросы раскрываются, и перед каждой строкой исходного кода появляются номера строк. По умолчанию -L. |

|  |  |
| --- | --- |
| -LISTLNO | подавляет перечисление номеров строк в выходном файле. Активно, только если также указана +L. По умолчанию +LISTLNO. |
| +LNO | помещает номер строки исходного кода перед первой соответствующей строкой U-кода в выходном файле. Выходные данные также указывают номер строки, с которой начинается каждый файл, созданный с помощью директивы #INCLUDE. Также четко обозначены вложенные #INCLUDE. По умолчанию -LNO. |
| -N | сразу за ним следует целое число, определяющее начальное значение для последовательно пронумерованных меток, генерируемых CC. По умолчанию - начальное значение два. |
| -OFILE | перенаправляет выходной файл на любой указанный диск под любым указанным именем файла. За ним сразу следует имя файла, которое будет использоваться для выходного файла. Между -OFILE и именем файла не должно быть пробелов. |
| +PRE | эквивалентно указанию +L -LISTNO -G. Создает только предварительно обработанный файл. Вырезание можно повторно запустить через CC. По умолчанию -PRE. |
| +S | печатает информацию о таблице символов по мере ввода каждого символа в таблицу символов. В основном это полезно для автора компилятора. По умолчанию -S. |
| +SILENT | подавляет вывод некоторой информации заголовка. Помогает уменьшить беспорядок на консоли. По умолчанию -SILENT. |
| +SUBM | обуславливает вывод, чтобы позволить отдельную сборку каждой функции C. Если этот переключатель указан как параметр CC, он также должен быть указан как параметр C2 (или C2I86). По умолчанию -SUBM. |
| +CASE | принимает ключевые слова без учета (верхнего/нижнего) регистра. Это относится как к ключевым словам препроцессора, так и к ключевым словам C. Полезно с терминалами, которые принимают только верхний регистр. Обратите внимание, что включаемый файл CBRACK.H содержит ключевые слова в верхнем регистре для распространенных токенов C, таких как { и }. Это полезно для клавиатур без нижнего регистра. По умолчанию -CASE. |
| -WBUFSIZ | сразу за ним следует десятичное целое число. Это целое число становится размером буфера для операций вывода. Чем больше это значение, тем быстрее выполняется компиляция. Меньшие размеры используют меньше памяти. Число 128 - это минимум для большинства систем. Если используется эта опция, выбранное значение будет напечатано на консоли. По умолчанию -WBUFSIZ1024. |

### Зависимые от машины опции командной строки CC

Три опции ниже могут использоваться для адаптации CC для конкретного целевого процессора (ЦП). Это единственные машинно-зависимые опции. Если значение, установленное для любого из этих параметров, не подходит для вашего процессора, компилятор не будет работать должным образом. (В дальнейшем jflag, int\_size и pad\_size относятся к именам переменных в компиляторе.)

|  |  |
| --- | --- |
| +J | устанавливает jflag равным единице (1). Каждый раз, когда один байт, представляющий char, передается функции как локальный параметр, он расширяется до int. Установка единицы (1) для jflag приводит к тому, что такой байт сохраняется в самом правом байте назначенного ему места хранения. Это правильная настройка для Z8000. Условие по умолчанию - установить равным нулю (0), что приводит к тому, что один и тот же байт, представляющий символ, сохраняется в крайнем левом байте назначенного ему места хранения. Это правильная настройка для процессоров серии 8080. |
| -I | сразу за ним следует целое число, определяющее размер int (int\_size) в байтах. По умолчанию - два байта. |
| -P | сразу же за ним следует целое число, указывающее соответствующий модуль (pad\_size) для всех целочисленных адресов. Поскольку каждый целочисленный адрес будет кратен значению pad\_size, каждый целочисленный адрес модуля значение pad\_size будет равно нулю ( 0). По умолчанию используется модуль единицы (1). |

Значения для трех вышеперечисленных параметров, необходимых для адаптации этого компилятора к конкретному целевому процессору, могут быть получены в одной операции с помощью одного из составных флагов, определенных ниже.

|  |  |
| --- | --- |
| +i8080 | устанавливает: jflag = 0, int\_size = 2, pad\_size = 1. Целевые процессоры - Intel 8080, 8085, 8086, 8088, 186, 188 и 286. |
| +z80 | устанавливает те же значения, что и выше. Целевой процессор - Zilog Z80. Примечание. Это опция CC, а не C2, поэтому мнемоника 8080 по-прежнему выводится C2. |
| +z8000 | устанавливает: jflag = 1, int\_size = 4, pad\_size = 2. Целевым процессором является Zilog Z8001. |
| +z8001 | устанавливает: jflag = 1, int\_size = 4, pad\_size = 2. Целевым процессором является Zilog Z8001. |
| +z8002 | устанавливает: jflag = 1, int\_size = 2, pad\_size = 2. Целевым процессором является Zilog Z8002. |

Условие по умолчанию идентично результату +i8080 или +z80.

### Параметры командной строки C2, COD2COD и C2I86

|  |  |
| --- | --- |
| +ASM | используется для обозначения того, что используется абсолютный ассемблер. Он включает в себя файл заголовка (по умолчанию: C2.RH или C2I86.RH) перед любым сгенерированным кодом и файл трейлера (по умолчанию: C2.RT или C2I86.RT) после любого сгенерированного кода. Эти файлы должны быть на диске по умолчанию. +ASM не используется под MS-DOS или PC-DOS. |
| -ASM | это значение по умолчанию, и оно указывает, что используется перемещающийся ассемблер. В CP/M вместо C2.RT используется C2.RTM, файл трейлера по умолчанию для перемещения ассемблеров, который добавляется в конец файла.) В MS-DOS или PC-DOS -ASM использует C2I86.RTB и C2I86.RTM в качестве файлов заголовка и трейлера соответственно для вывода определений кода. |
| -BUFSIZ | сразу за ним следует десятичное целое число. Это целое число становится размером буфера для операций ввода-вывода. Чем больше это значение, тем быстрее выполняется компиляция. Меньшие размеры используют меньше памяти. Число 128 - это минимум для большинства систем. Этот размер действует как для входного, так и для выходного буферов (см. -WBUFSIZ ниже). Если используется эта опция, выбранное значение будет напечатано на консоли. По умолчанию -BUFSIZ1024. |
| +CO | принудительно выводит на консоль, где можно просмотреть вывод. По умолчанию выводится в filename.ASM. |
| -ENT<keyword> | определяет ключевое слово, используемое для объявления точки входа метки для перемещающих ассемблеров. (Примечание: нет пробела между -ENT и ключевым словом.) Значение по умолчанию -ENTENTRY. |
| -EXT<keyword> | определяет ключевое слово, используемое для объявления внешней метки для перемещающих ассемблеров (Примечание: нет пробела между -EXT и ключевым словом). По умолчанию -EXTPUBLIC. |
| -G | отключает генерацию кода ассемблера, по умолчанию используется генерация выходного кода ассемблера. |
| +L | помещает каждую строку U-кода в выходной файл ассемблерного кода в качестве комментария. Если выполняется оптимизация, строки U-кода группируются по функциям; в противном случае они возникают непосредственно перед кодом, генерируемым для строки. По умолчанию -L. |
| +MSDOS | инструктирует C2I86 выводить код, совместимый с ассемблером MS-DOS/PC-DOS. |
| -O | отключает оптимизацию. Выполняется только перевод U-кода в ассемблерный код. По умолчанию выполняется оптимизация. |
| +OC | отслеживает процесс оптимизации, записывая информацию в выходной файл с указанием каждого внесенного изменения кода. Эта опция замедлит выполнение C2 или COD2COD и значительно увеличит объем вывода. Особо полезен только автору оптимизатора. По умолчанию -OC. |
| -OFILE | перенаправляет выходной файл на любой указанный драйвер под любым указанным именем файла. За ним сразу следует имя файла, которое будет использоваться для выходного файла. Между -OFILE и именем файла не должно быть пробелов. |
| -ORG | за которым сразу следует десятичное целое число, заставляет выходной код быть расположенным с указанным значением. По умолчанию 256, если указан +ASM; в противном случае вообще нет ORG. |
| +PRGL | вызывает отображение имени оптимизируемой в данный момент функции на консоли в качестве указания того, где находится C2 или COD2COD в процессе ее выполнения. По умолчанию -PRGL. |
| -Q | сразу же за ним следует целое число, задающее уровень оптимизации. По умолчанию используется полная оптимизация. (В настоящее время не реализована.) |
| -RH<filename> | заставляет оптимизатор использовать указанный файл в качестве файла заголовка времени выполнения. (Примечание: между -RH и именем файла нет пробела.) По умолчанию - C2.RH (C2I86.RH в CP/M-86), если указан +ASM. Для -ASM в CP/M не добавляется никаких файлов; C2I86.RTB используется под MS-DOS. |
| -RT<filename> | заставляет оптимизатор использовать указанный файл в качестве файла трейлера времени выполнения. (Примечание: между -RT и именем файла нет пробела.) По умолчанию C2.RT (C2I86.RT в CP/M-86), если указан +ASM; для -ASM, C2.RTM (C2I86.RTM под MS-DOS). |
| +T | генерирует и вставляет код в выходной файл, который во время выполнения заставляет отображать имя каждой функции каждый раз, когда она вводится. Эта опция, полезная при отладке, позволяет программисту отслеживать поток управления в программе. (В настоящее время не реализовано.) По умолчанию -T. |
| -WBUFSIZ | сразу за ним следует десятичное целое число. Это целое число становится размером буфера для операций вывода. Чем больше это значение, тем быстрее выполняется компиляция. Меньшие размеры используют меньше памяти. Число 128 - это минимум для большинства систем. Если этот параметр используется, выбранное значение будет напечатано на консоли.  По умолчанию -WBUFSIZ1024. |
| -X | вызывает генерацию чуть меньшего размера кода за счет скорости его выполнения. Произведенная оптимизация способствует повышению эффективности использования пространства. По умолчанию эффективность использования пространства меняется на эффективность скорости. Разница относительно небольшая, потому что большинство оптимизаций положительно сказываются как на скорости, так и на экономии места. (В настоящее время не реализована.) |
| +Z | сразу за ним следует строка символов, которую C2 (C2I86 в CP/M-86 и MS-DOS) использует в качестве префикса для всех генерируемых им меток. Префикс 'C' используется по умолчанию (+ZC). |

## Директивы препроцессора компилятора

Все файлы, вводимые в компилятор SuperSoft C, проходят через препроцессор, который сканирует их на наличие строк, начинающихся с '#'. Такие строки, которые сами по себе не компилируются, а направляют процесс компиляции, называются директивами препроцессора компилятора. Эти строки заканчиваются символами новой строки, а не точками с запятой. Директивы препроцессора, поддерживаемые в SuperSoft C, следующие:

|  |  |
| --- | --- |
| **#define**  **#endif**  **#else**  **#include**  **#ifdef** | **#ifndef**  **#line**  **#undef**  **#asm**  **#endasm** |

Каждая из этих директив синтаксически независима от остального синтаксиса языка Си и может появляться в любом месте программы. Действие всех директив распространяется до конца компиляции. Их действие и применение описаны в нижеследующих пунктах.

### Директива #define

Если строка формы

**#define IDENTIFIER TOKENSTRING**

находится во входных данных источника, она заставляет препроцессор заменять все последующие вхождения IDENTIFIER на указанный в TOKENSTRING. (Токен - это лексическая единица языка программирования, такая как идентификатор, оператор, ключевое слово и т. д.) Это простейшая форма подстановки макросов. Если #DEFINE идентификатор встречается в двойных или одинарных кавычках, он не будет заменен. Каждая замещающая строка будет повторно сканироваться для других идентификаторов #DEFINE, что позволяет вложить #DEFINE директивы. Переопределение идентификатора приводит к ошибке компиляции. Параметризованный #DEFINE не поддерживается.

Чаще всего эта директива используется для определения набора символьных констант в заголовке программы для использования в следующих функциях (гораздо лучше, чем вставка буквальных констант в операторы программы). Сокращенный пример такого использования:

**#define BSIZ 0x103**

**char Buf1[BSIZ], Buf2[BSIZ);**

**main() {**

**register int f;**

**for (i=0;i++<BSIZ;) {**

**Buf1[i] = Buf2[i] = 0;**

**}**

**}**

Подстановка макросов с помощью директивы #define имеет много других применений. Дополнительные примеры и информацию см. На страницах 12, 86 и 207 в книге Кернигана и Ричи.

### Директива #line

Директива line устанавливает номер исходной строки, используемый компилятором. Это будет содержать номер строки, напечатанный компилятором при возникновении ошибки. Форма имеет следующий вид:

**#line LINENUMBER**

где LINENUMBER - константа.

### Директива #include

Если строка формы

**#include "drivename:filename"**

или строка вида:

**#include <drivename:filename>**

вводится компилятору, он заставляет препроцессор заменять эту строку всем содержимым файла с таким именем на указанном диске. Если препроцессор не может найти файл, он будет искать его на текущем диске. В будущем (то есть не в текущей версии) препроцессор будет искать указанный файл на всех дисках, которые, как он знает, доступны (все диски, упомянутые до этого момента), если директива #INCLUDE <filename> является директивами #INCLUDE найденный в уже вложенном файле она будет обработан таким же образом. Следовательно, используется вложение. Возможен любой файл с директивами #INCLUDE. Примеры и информацию см. На страницах 86, 143 и 207 в Кернигана и Ричи.

### Директивы #IF, #IFDEF, #IFNDEF, #ELSE и #ENDIF

Эти директивы выполняют условную компиляцию. Общая форма:

**#CONDITIONAL**

**истинный код**

**#else**

**ложный код**

**#endif**

где #CONDITIONAL - одно из #IF, #IFDEF или #IFNDEF. #ELSE является необязательным. То есть форма может быть:

**#CONDITIONAL**

**истинный код**

**#endif**

Если #CONDITIONAL считается истинным , то строки, показанные как истинный код, компилируются, а строки с обозначенные как ложный код игнорируются. Если #CONDITIONAL считается ложью, то компилируются строки с показанные как ложный код, а строки с истинныйм кодом игнорируются.

Директива #IF имеет вид:

**#IF CONSTANT**

Поле CONSTANT должно быть константой. Если это так, то #IF считается истинным.

Директива #IFDEF имеет вид:

**#ifdef IDENTIFIER**

IDENTIFIER должен быть идентификатором в стиле #DEFINE. Если IDENTIFIER был ранее определен через #DEFINE, то #IFDEF считается верным.

**#ifndef IDENTIFIER**

IDENTIFIER должен быть идентификатором в стиле #DEFINE. Если IDENTIFIER был ранее определен с помощью #DEFINE, то #IFNDEF считается ложным.

### Директивы #ASM и #ENDASM

Эта особенность SuperSoft C позволяет вставлять строки ассемблерного кода непосредственно в файл исходного кода C. Это могло бы выглядеть следующим образом:

**...**

**putcar('y')**

**#asm**

**mvi a,88**

**call output**

**#endasm**

**crlf()**

**...**

Все строки ассемблера между #ASM и #ENDASM проходят через проходы компилятора без изменений и включаются в соответствующие места в окончательном выходном файле. Внутри функции C строки будут выполняться так, как они встречаются. Вне функции строки должны содержать метку и должны быть явно вызваны для выполнения. Типы строк на языке ассемблера, которые выводятся за пределы функции C, являются директивами и эквивалентами. Никакие включенные строки ассемблера не оптимизируются и не проверяются на наличие ошибок. Следует избегать меток, начинающихся с 'C' во вставленном коде, поскольку метки с этим префиксом генерируются компилятором и могут привести к дублированию меток при ассемблировании программы.

Включенные строки ассемблера можно использовать где угодно в исходном файле C. Если они возникают вне функции C или если параметр -O (не оптимизировать) используется на проходе C2 или COD2COD, то количество включаемого языка ассемблера не ограничено. В противном случае количество ассемблера будет ограничено оптимизацией для каждой процедуры C2 или COD2COD.

Следующие параметры предполагаются C2 в CP/M-80 и будут влиять на то, как используются включенные строки ассемблера:

**.RADIX 10 (decimal radix),**

**.13233 (838C! mnemonics),**

**CSEG (code segment).**

На глобальные объекты C можно ссылаться во включенном ассемблере (а также на связанном ассемблере), а на символы ассемблера может ссылаться C. В настоящее время глобальные объекты C будут отображать символы ассемблера под тем же именем. Так будет не всегда. В следующей версии компилятора к символу C будет добавлен дополнительный постфиксный символ, чтобы он соответствовал глобальному значению ассемблера. Также у STATIC не будет легко доступного символа ассемблера.

C использует эти регистры:

первичный регистр (текущее подвыражение),

вторичный регистр (предыдущее подвыражение),

указатель стека,

регистровая переменная,

базовый указатель (только в некоторых реализациях).

C ожидает постоянного сохранения регистровой переменной; первичный регистр, который должен быть установлен на возвращаемые значения при возврате функции; а первичный и вторичный регистры должны быть сохранены при вычислении любого подвыражения. Указатель стека также должен сохраняться при вызовах функций и включенном языке ассемблера. Вторичный регистр более временный: не ожидается, что он будет сохраняться при вызовах функций. Однако ожидается, что он будет сохранен в большинстве подвыражений и во время векторизации в случае переключения. Обычно нежелательно или полезно навязывать включенный язык ассемблера внутри подвыражения C. Пример включения языка ассемблера в подвыражение C, которое может вызвать проблемы:

**a =**

**#asm**

**mvi h,9**

**#endasm**

**b;**

При вызове кода C из ассемблера или любого другого языка лучше не предполагать, что какие-либо регистры сохраняются кодом C, хотя некоторые регистры сохраняются некоторыми генераторами кода SuperSoft C. При вызове языка ассемблера из C или при "отказе", включая строки языка ассемблера, обязательно сохраните регистровую переменную и указатель стека. Другие (в первую очередь, первичные) регистры, возможно, придется сохранить на некоторых машинах. Например, регистры DS, CS и SS должны быть сохранены на 8086. В следующей таблице суммируется назначение регистров C.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Регистр** | **8080/z80** | | **8086/286** | | **z8002** | | **z8001** | |
| первичный | | HL | | BX | | R4 | | RR4 |
| вторичный | | DE | | SI | | R5 | | RR6 |
| указатель стека | | SP | | SP | | R15 | | RR14 |
| регистровая переменная | | BC | | DI | | R8 | | RR8 |

### Директива #UNDEF

Эта директива имеет форму

**#undef IDENTIFIER**

Она заставляет препроцессор забыть названный идентификатор.

## Использование функций list

Компилятор SuperSoft C может обрабатывать функции без заранее определенного количества аргументов. Это делается с помощью дополнительного атрибута функции, называемого list (список). Стандартные функции библиотеки SuperSoft C включают следующие стандартные функции list: printf, scanf, fprintf, fscanf, sprintf и sscanf. Работа этих функций list прозрачна для пользователя: вам не нужно делать ничего особенного, чтобы воспользоваться преимуществами этих встроенных функций.

Однако, если вы хотите использовать свои собственные функции list, вам нужно знать, как объявляется атрибут list и как передаются параметры. Для правильного выполнения созданных вами функций list их необходимо объявить до того, как они впервые будут использованы вашей программой, как показано ниже:

**TYPE listf(.) ;**

где listf - это произвольная функция list, а TYPE - это объявление возвращаемого значения для listf, которое, конечно же, должно соответствовать всем другим объявлениям для listf. Это объявление приводит к тому, что при каждом вызове этой функции последним аргументом в стеке параметров будет счетчик аргументов. Если это объявление появляется в верхней части вашей программы как глобальное внешнее, оно должно появиться в вашей программе только один раз.

Тело функции listдолжно собирать свои аргументы особым образом, чтобы учесть измененный стек параметров. Вы можете посмотреть в теле одной из вышеуказанных функций специальное кодирование, необходимое для использования дополнительной информации, доступной через стек параметров. Атрибут list позволяет функциям C, которые ожидают определенного количества аргументов, защищать себя от передачи неправильного количества аргументов, что позволяет кодировать очень надежные программы. Например, можно обнаружить следующий плохо сформированный вызов функции:

**fprintf("Is") ;**

Эта способность обнаружения является особенностью, которой обладают немногие компиляторы языка Си. В большинстве компиляторов языка Си приведенный выше код приведет к возникновению неопределенных (и потенциально катастрофических) операций ввода-вывода.

Тело функции list должно выглядеть примерно так:

**pf (nargs)**

**int nargs[];**

**{**

**unsigned cnt;**

**int \* p;**

**cnt \* nargs;**

**if (cnt < MINIMUM || cnt > MAXIMUM)**

**return;**

**p = Xrev(&nargs); /\* lib arg reversal fn \*/**

**/\***

**\* at this point:**

**\* cnt is the count of passed parameters**

**\* p[0] is the leftmost parameter**

**\* p[1] is the 2nd leftmost parameter**

**\* p[2] is the 3rd leftmost parameter**

**\* ... and so on up to p[cnt-l]**

**\*/**

**}**

# Глава 3 Функции стандартной библиотеки SuperSoft C

Стандартные библиотечные функции, поставляемые с компилятором SuperSoft C, определены в следующих файлах:

**в CP/M-80**

|  |  |  |
| --- | --- | --- |
| C2.RH | : | заголовочный файл библиотеки времени выполнения |
| C2.RT | : | файл трейлера времени выполнения |
| C2.RTM | : | трейлера времени выполнения для RMAC и M80 |
| C2PRE.ASM | : | источник заголовка времени выполнения для L80 |
| C2POST.ASM | : | исходный код трейлера времени выполнения для L80 |
| MDEP.C | : | #ASM/#ENDASM версия среды выполнения |
| CBRACK.H | : | определения для клавиатур без нижнего регистра |
| CUSTOMIZ.H | : | параметры времени компиляции библиотеки C |
| STDIO.H | : | заголовок функций стандартного ввода/вывода |
| STDIO.C | : | функции стандартного ввода/вывода |
| ALLOC.C | : | функции распределения динамической памяти |
| CRUNT2.C | : | части среды выполнения на языке C |

|  |  |  |
| --- | --- | --- |
| LONG.C | : | длинные целочисленные функции |
| DOUBLE.C | : | функции с плавающей запятой удвоенной точности |
| BCD80.C | : | поддержка языка ассемблера для DOUBLE.C |

|  |  |  |
| --- | --- | --- |
| FUNC.C | : | вспомогательные функции |
| FORMATION.C | : | printf, scanf и др. функции |

**в CP/M-86**

|  |  |  |
| --- | --- | --- |
| C2I86.RH | : | заголовочный файл библиотеки времени выполнения |
| C2I86.RT | : | файл трейлера времени выполнения |
| MDEP.C | : | #ASM/#ENDASM версия среды выполнения |
| CUSTOMIZ.H | : | параметры времени компиляции библиотеки C |
| CBRACK.H | : | определения для клавиатур без нижнего регистра |
| STDIO.H | : | заголовок функций стандартного ввода/вывода |
| STDIO.C | : | функции стандартного ввода/вывода |
| ALLOC.C | : | функции распределения динамической памяти |
| CRUNT2.C | : | части среды выполнения на языке C |

|  |  |  |
| --- | --- | --- |
| LONG.C | : | длинные целочисленные функции |
| DOUBLE.C | : | функции с плавающей запятой удвоенной точности |

|  |  |  |
| --- | --- | --- |
| FUNC.C | : | вспомогательные функции |
| FORMATION.C | : | printf, scanf и др. функции |

**в MS-DOS и PC-DOS**

|  |  |  |
| --- | --- | --- |
| C2I86.RTB | : | заголовочный файл библиотеки времени выполнения |
| C2I86.RTM | : | файл трейлера времени выполнения |
| C2PRE.ASM | : | источник заголовка времени выполнения для LINK |
| C2POST.ASM | : | исходный код трейлера времени выполнения для LINK |
| MDEP.C | : | #ASM/#ENDASM версия среды выполнения |
| CBRACK.H | : | определения для клавиатур без нижнего регистра |
| CUSTOMIZ.H | : | параметры времени компиляции библиотеки C |
| STDIO.H | : | заголовок функций стандартного ввода/вывода |
| STDIO.C | : | функции стандартного ввода/вывода |
| ALLOC.C | : | функции распределения динамической памяти |
| CRUNT2.C | : | части среды выполнения на языке C |
| FUNC.C | : | вспомогательные функции |
| FORMATION.C | : | printf, scanf и др. функции |
| LONG.C | : | длинные целочисленные функции |
| DOUBLE.C | : | функции с плавающей запятой удвоенной точности |

Во время второго прохода (или третьего прохода, в случае CP/M-86 и MS-DOS) компилятор автоматически включает в вашу программу определенный предварительно скомпилированный, предварительно оптимизированный ассемблерный код. Этот код содержится в файлах библиотеки времени выполнения компилятора. Остальные файлы содержат определения остальных функций стандартной библиотеки в исходном коде SuperSoft. Существуют различные методы объединения ваших программ с функциями библиотеки. Они описаны в [*главе 4*](#_Глава_4_Объединение_1).

## Описание функций стандартной библиотеки SuperSoft C

Остальная часть этой главы будет состоять из описаний каждой из функций стандартной библиотеки SuperSoft C в алфавитном порядке. Страницы языка программирования C Кернигана и Ричи, на которые ссылается большинство функций, не предоставляют подробностей о нашей конкретной реализации этой функции, а вместо этого даются как руководство по использованию такой функции в целом. Поскольку большинство функций стандартной библиотеки предоставляется в исходном коде SuperSoft C, а остальные - в коде сборки, вы можете узнать подробности нашей реализации, изучив этот код.

Представляя каждую из функций ниже, мы приняли своего рода сокращенное обозначение типов данных, возвращаемых и передаваемых в качестве аргументов каждой функции. Первые несколько строк, которые появляются перед описанием каждой функции, являются первыми строками ее определения в SuperSoft C, а именно объявления самой функции и ее аргументов. Тип функции - это тип возвращаемого значения. Если тип функции не указан, она не возвращает никакого определенного значения. Всегда будет указан тип каждого из его аргументов.

Файл STDIO.H содержит операторы #DEFINE, устанавливающие следующие предопределенные символические возвращаемые значения (фактические возвращаемые числовые значения показаны в скобках):

TRUE (1)

FALSE (0)

SUCCESS (0)

ERROR (-1)

NULL (0)

Файл STDIO.H также содержит операторы #DEFINE, устанавливающие различные типы данных, которые используются библиотечными функциями. К ним относятся:

|  |  |  |
| --- | --- | --- |
| **Тип данных** | **Определение** | **Объяснение** |
| BOOL | char | TRUE или FALSE |
| RESULT | int | SUCCESS или FAILURE |
| FILE | struct filedesc | файловый дескриптор |
| FILESPEC | char \* | имя файла |
| STREAM | file \* | буферизованный файловый дескриптор |
| TYPE |  | произвольный или переменный тип |

STDIO.C содержит #INCLUDE STDIO.H, и многие из функций, определенных в STDIO.C, будут возвращать эти значения.

## Функции сгруппированы в соответствии с использованием

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Доступ к системе** | | | **Управление потоком** | | | **Ввод-вывод на консоль** | |
| bdos | inp | | exec | setjmp | | getchar | putchar |
| bios | outp | | execl | setexit | | gets | putdec |
| exit | inp16 | | longjmp | sleep | | kbhit | puts |
| ccexit | outp16 | | reset | wait | | scanf | printf |
| ccall | cpmver | | pause | | | ugetchar | |
| ccalla | xmain |  | | | | | |
| comlen | lock |
| comline | nice |
| **Буферизованный ввод-вывод файлов** | |  | **Прямой ввод-вывод файлов** | |  | **Общие файлы** | |
| fclose | fread | | close | rtell | | fabort | rename |
| fflush | fscanf | | creat | seek | | unlink | link |
| fgetc | fwrite | | open | tell | | perror | errno |
| fgets | getc | | otell | write | | isfd | isatty |
| fopen | getw | | read | access | | mktemp |  |
| fprintf | putc | | chmod |  | | | |
| fputc | putw | | | | | | |
| fputs | ungetc | | **Инициализация памяти** | | | **Цифровые** | |
| fdopen | freopen | | | | | | |
| clearerr | pgetc | | initb | peek | | abs | absval |
| pputc | ferror | | initw | poke | | max | min |
| get2b | put2b | | movmem | getval | | rand | srand |
| fileno | | | setmem | | | double | long |
|  | | | | | | assert |  |
| **Обработка строк** | | |  | | | | |
| atoi | xrprintf | |
| index | xrscanf | | **Динамическое распределение памяти** | | | | |
| isprint | rindex | |  | | | | |
| isspace | sprintf | | alloc | malloc |  | | |
| ispunct | sscanf | | calloc | realloc |
| iswhite | strcat | | brk | sbrk |
| isalnum | strcmp | | ubrk | wrdbrk |
| isalpha | strcpy | | envbrk | free |
| isascii | streg | | xrev | isheap |
| iscntrl | strlen | | topofmem |  |
| isdigit | strncat | |  | | | | |
| isupper | strncmp | |
| islover | strncpy | |
| isnumeric | substr | |
| qsort | tolower | |
| swab | toupper | |

## Список функций

**ABS**

int abs(i)

int i;

Функция abs возвращает абсолютное значение i (-i, если i меньше нуля, в противном случае i). Эта функция недоступна в MS-DOS или CP/M-86 из-за конфликта ключевых слов ассемблера. См. absval ниже.

**ABSVAL**

int absval(i)

int i;

Функция absval возвращает абсолютное значение i (-i, если i меньше нуля, в противном случае i). См. abs выше.

**ACCESS**

BOOL access(filename,mode)

FILESPEC filename;

unsigned mode;

Функция access возвращает TRUE (1), если файл доступен для данного режима; FALSE (0) в противном случае.

**ALLOC**

char \*alloc(n)

unsigned n;

Функция alloc выделяет непрерывную область памяти длиной n. Каждый выделяемый ей блок начинается с четного адреса. См. ее идентичный близнец malloc.

**ASSERT**

BOOL ASSERT(b)

BOOL b;

Функция assert выводит на консоль «Assertion failed\n» и завершает работу, если b имеет значение FALSE; в противном случае assert просто возвращается.

**ATOI**

int atoi(s)

char \*s;

Функция atoi возвращает десятичное целое значение, соответствующее нулевой строке ASCII, на которую указывает s. Если эта строка содержит символы, отличные от ведущих табуляций, пробелов и знака минус (все необязательные), за которыми следуют последовательные десятичные цифры, atoi возвращает 0 (см. K&R, стр. 39, 58).

**BDOS**

int bdos(fn, parm)

int fn, parm;

Функция bdos позволяет пользователям включать прямые вызовы функций BDOS в программы, написанные на SuperSoft C. Программы, использующие функцию bdos, не всегда будут переносимыми. Например, вызов функции BDOS 7 в CP/M-80 и CP/M-86 представляет собой "Получить байт ввода-вывода", но в MS-DOS тот же номер функции представляет собой "прямой консольный ввод". С другой стороны, вызов функции BDOS 9, "Печать строки", представляет одну и ту же функцию во всех трех системах. Для получения дополнительной информации пользователям следует обратиться к руководству по системному интерфейсу.

В CP/M-80 bdos загружает машинный регистр C с помощью fn и пару машинных регистров DE с помощью parm, затем вызывает ячейку 5. Она возвращает (как целое число) один байт, идентичный содержимому регистра A, за исключением вызовов функций BDOS 12, 24, 27, 29 и 31, когда он возвращает слово, идентичное содержимому пары регистров HL.

В CP/M-86 bdos загружает машинный регистр CX с помощью fn и машинный регистр DX с помощью parm, затем выполняет инструкцию INT 224. Она возвращает (как целое число) один байт, идентичный содержимому регистра AL, за исключением вызовов функций BDOS 12, 24, 27, 29 и 31, когда она возвращает слово, идентичное содержимому регистра BX. Кроме того, вызовы 27 и 31 устанавливают для глобальной переменной ESS значение, возвращаемое в регистре ES; вызов 52 возвращает слово, идентичное содержимому регистра ES, а также устанавливает для глобальной переменной BXX значение, возвращаемое в BX; а вызов 59 возвращает слово, идентичное содержимому регистра AX. Для получения информации о вызове 50 см. BIOS, который описан позже.

В MS-DOS bdos загружает машинный регистр AH с помощью fn и машинный регистр DX с помощью parm, а затем выполняет инструкцию INT 0x21. Она возвращает (как целое число) один байт, идентичный содержимому регистра AL, за исключением вызова функции BDOS 6, когда он возвращает AL, плюс 0x400, если установлен нулевой флаг. Другие исключения в MS-DOS: вызовы функций 12 и с 31 по 40 возвращают недопустимые результаты; вызовы 24 и с 28 по 32 не определены в этой системе.

Сводка исключений BDOS для CP/M, CP/M-86 и MS-DOS приведена ниже:

|  |  |  |  |
| --- | --- | --- | --- |
| **CP/M-80** **Все возвращают A, кроме:** | | | |
| Вызов функции BDOS | Возвращает | |  |
| 12 (возврат номера версии) | | HL |  |
| 24 (возврата вектора обращения) | | HL |  |
| 27 (получить адрес (Alloc)) | | HL |  |
| 29 (получить вектор R/O) | | HL |  |
| 31 (получить адрес (параметров диска)) | | HL |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CP/M-86** **Все возвращают AL, кроме:** | | | | |
| Вызов функции BDOS | Возвращает | | Устанавливает | |
| 12 (возврат номера версии) | | BX | |  |
| 24 (возврата вектора обращения) | | BX | |  |
| 27 (получить адрес (Alloc)) | | BX | | ESS |
| 29 (получить вектор R/O) | | BX | |  |
| 31 (получить адрес (параметров диска)) | | BX | | ESS |
| 50 (прямой вызов BIOS) | | см. BIOS | |  |
| 52 (установить начало сегмента DMA) | | ES | | ESS |
| 59 (загрузить программу) | | AX | |  |

|  |  |
| --- | --- |
| **MS-DOS** **Все возвращают AL, кроме:** | |
| Вызов функции BDOS | Возвращает |
| 6 (прямой ввод-вывод консоли) | AL и 0x400, если установлен нулевой флаг |
| 12 (Ввод символов с очисткой буфера) | Неверная функция |
| 24, 28-32 | Эти функции не определены в MS-DOS |
| 33-40 | Неверная функция |

ПРИМЕЧАНИЕ. Значение, которое возвращает bdos, не расширяется по знаку.

**BIOS**

|  |  |
| --- | --- |
| **CP/M-80** | **CP/M-86** |
| int bios(jmpnum, bc, de)  int jmpnum;  int bc, de; | int bios(jmpnum, cx, dx)  int jmpnum;  int cx, dx; |

bios позволяет пользователям CP/M включать прямые вызовы BIOS в программы, написанные на SuperSoft C. Программы, вызывающие BIOS, не будут переноситься за пределы систем CP/M. bios устанавливает машинную регистровую пару BC (CX в CP/M-86) на значение, указанное в BC (CX в CP/M-86), а регистровую пару BC (CX в CP/M-86) на значение, указанное в DE (DX под CP/M-86) и инициирует другой вызов BIOS, передав управление в точку входа вектора перехода BIOS, указанную jmpnum. Эта точка входа может быть указана численно или символически.

Соответствующие мнемонические символы или имена символов для каждой точки (как указано в Руководстве по изменениям CP/M и Справочном руководстве по системе CP/M-86, предоставленных Digital Research, Inc.) и числовое значение для каждой точки входа приведены ниже:

**Точки входа в CP/M-80**

|  |  |  |
| --- | --- | --- |
| BOOT 0 | PUNCH 6 | SETDMA 12 |
| WBOOT 1 | READER 7 | READ 13 |
| CONST 2 | HOME 8 | WRITE 14 |
| CONIN 3 | SELDSK 9 | LISTST 15 |
| CONOUT 4 | SETTRK 10 | SECTRAN 16 |
| LIST 5 | SETSEC 11 |  |

В CP/M, если jmpnum имеет значение SELDSK (9) или SECTRAN (16), bios возвращает значение, оставшееся в регистре HL после выполнения вызова BIOS, в противном случае она возвращает значение, оставшееся в регистре A без расширения знака (то есть как значение от 0 до 255).

**Точки входа в CP/M-86**

|  |  |  |
| --- | --- | --- |
| BOOT 0 | READER 7 | WRITE 14 |
| WBOOT 1 | HOME 8 | LISTST 15 |
| CONST 2 | SELDSK 9 | SECTRAN 16 |
| CONIN 3 | SETTRK 10 | SETDNAB 17 |
| CONOUT 4 | SETSEC 11 | GETSEGB 18 |
| LIST 5 | SETDMA 12 | GETIOB 19 |
| PUNCH 6 | READ 13 | SETIOBB 20 |

В CP/M-86, если jmpnum имеет значение SELDSK (9), SECTRAN (16) или GETSEGB (18), bios возвращает значение, оставшееся в регистре EX после выполнения вызова BIOS; в противном случае возвращается значение, оставшееся в регистре AL. Кроме того, SELDSK и GETSEGB устанавливают для глобальной переменной ESS значение, значение, возвращаемое в регистре ES.

**BRK**

RESULT brk(p)

char \*p;

Функция brk устанавливает имя внешней переменной CCEDATA равным байту памяти, на который указывает p, и возвращает CCEDATA, что эквивалентно переданному значению указателя. CCEDATA изначально устанавливается в байт, следующий сразу за байтом в области внешних данных вашей программы. Поскольку CCEDATA используется в качестве базового значения другими функциями распределения динамической памяти, это правильно инициализирует эти функции в вашей программе. Вам почти никогда не понадобится вызывать brk в вашей собственной программе. Функция brk возвращает ERROR, если не может выполнить прерывание.

**CALLOC**

char \*calloc(nthings, sizeofthings)

unsigned nthings, sizeofthings;

Выделяет непрерывную область памяти, длина которая равна произведению nthings и sizeofthings. Каждый выделенный блок начинается с четного адреса. Байты блока содержат нулевое (0) значение. См. malloc.

**CCALL**

BOOL ccall(adr, hl, a, bc, de)

char \*adr, a;

int hl, bc, de;

Функция ccall устанавливает машинные регистры HL, A, BC и DE на значения, указанные в hl, a, bc и de соответственно, и вызывает подпрограмму ассемблера, начинающуюся с adr. Функция ccall возвращает значение, присутствующее в регистре HL после выполнения подпрограммы. Программы, вызывающие ccall, не будут переноситься за пределы процессоров серии 8080.

**CCALLA**

BOOL ccalla(adr, hl, a, bc, de)

char \*adr, a;

int hl, bc, de;

Функция ccalla идентична ccall, за исключением того, что она возвращает значение, имеющееся в регистре A после выполнения подпрограммы. Программы, вызывающие ccalla, также не будут переноситься за пределы ЦП серии 8080.

В качестве примера использования функции рассмотрим следующее:

**int bios (c, de)**

**char c;**

**int de;**

**{**

**return ccalla (5, 0, 0, c, de);**

**}**

Перечисленная выше функция C явно является реализацией функции BDOS с точки зрения ccalla. (BDOS реализуется иначе.) Как ccall, так и ccalla могут также использоваться для вызова подпрограмм языка ассемблера, созданных вами.

**CCEXIT**

BOOL ccexit(i)

int i;

Возврат в операционную систему. Параметр i - это код возврата. Параметр i должен быть 0, если программа завершена успешно. В противном случае i должен быть от 1 до 255. (Эти значения могут быть произвольно присвоены кодам ошибок.) Функция ccexit закрывает все открытые файлы.

**CHMOD**

SUCCESS chmod(mode)

unsigned mode;

Устанавливает режим файла. Если установлен бит 0x80, устанавливает R/O для CP/M и CP/M-86.

**CLEARERR**

clearerr(fd)

STREAM fd;

Очищает состояние ошибки файла. В настоящее время существует только одна глобальная переменная состояния ошибки: errno (описана позже).

**CLOSE**

close(fd)

FILE fd;

Закрывает файл, указанный указателем fd на его файловый дескриптор. Функция close возвращает SUCCESS (0), если указанный файл был успешно закрыт. Функция close возвращает ERROR (-1) и не закрывает файл, если: (1) fd не указывает на допустимый файловый дескриптор или (2) файл не может быть закрыт из-за ошибки на уровне операционной системы.

Функция close не помещает символ конца файла (EOF) в буфер (если он назначен файлу) и выполняет вызов fflush. Если вы вызываете close для открытого файла, буферизованный вывод через fopen, без предварительного вызова fflush, будут потеряны все данные, оставшиеся в буфере ввода-вывода этого файла. (См. K&R, стр. 163).

**CODEND**

char \*codend()

Функция codend возвращает указатель на байт, следующий сразу за концом кода для корневого сегмента вашей программы. Если вы не заново создали область внешних данных своей программы, возвращаемое значение codend будет указывать на начало этой области. (В настоящее время не реализована.)

**COMLEN**

int comlen()

Возвращает длину командной строки. Доступна только в системах, в которых командная строка доступна в виде массива символов.

**COMLINE**

char \*comline()

Возвращает адрес командной строки. Доступна только в системах, в которых командная строка доступна в виде массива символов.

**CPMVER**

int cpmver()

Возвращает значение, присутствующее в регистре HL, после выполнения вызова функции BDOS номер 12 (см. Руководство по интерфейсу CP/M, предоставленное Digital Research Inc.). Это возвращаемое значение, которое не расширяется знаком, равно 0, если вызывающая программа работает под версиями CP/M, выпущенными до версии 2.0, 0x0020, если она работает под CP/M версии 2.0, в диапазоне от 0x0021 до 0x002F в версиях после 2.0 и 0x0100 в MP/M. Эта функция полезна при написании программ на C для работы под CP/M или MP/M, независящих от номера версии. Программы, вызывающие cpmver, не будут переноситься за пределы CP/M, CP/M-86 и MP/M.

**CREAT**

FILE \*creat(fspec, mode)

FILESPEC fspec;

unsigned mode;

Создает файл на диске со спецификацией файла, указанной в fspec и открывает его для вывода. Если для значения mode (режима) установлен бит 0x80, то создается файл с возможностью чтения и записи, в противном случае создается файл только для чтения. Любой существующий файл с такой же спецификацией будет удален.

Функция creat, в случае успеха, возвращает указатель на допустимый файловый дескриптор для указанного файла. Вы должны сохранить этот указатель для последующего использования в вашей программе. creat возвращает ошибку (-1) и не создает и не открывает ни одного файла, если: (1) недостаточно памяти пользователя для хранения нового файлового дескриптора, (2) Данная спецификация файла недействительна или (3) файл не может быть создан и открыт из-за ошибки на уровне операционной системы (см. K&R, стр. 162).

**DOUBLE арифметика**

Файл DOUBLE.C содержит арифметический пакет с плавающей запятой двойной точности. Эти функции выполняют арифметику для чисел BCD длиной 3 байта. Все переменные, используемые этим пакетом, должны быть объявлены DOUBLE или LONG FLOAT. Функции передают адрес каждого аргумента. Обычно функции возвращают адрес назначения так что вызовы могут быть вложены следующим образом:

Badd (&d, Bmul(&a, &b, &c) , &e)

В этом примере d=b\*c+e.

См. [*Приложение G*](#_Приложение_G_Функции_1) для описания этих функций.

**ENDEXT**

char \*endext()

Возвращает указатель на байт, следующий сразу за последним байтом в области внешних данных вашей программы. Это значение должно быть идентично начальному значению CCEDATA. (В настоящее время не реализована.)

**ERRNO**

int errno

Не функция, а внешняя переменная, которая будет присвоен код ошибки всякий раз, когда произойдет ошибка ввода-вывода. Обратите внимание, что она не очищается автоматически, если ошибка не возникает. Нулевого сообщения об ошибке нет, поэтому очистка errno - это общепринятый способ "предустановить" его для сбора значений ошибок. Функция perror, описанная ниже, выводит на консоль сообщение об ошибке с ненулевой ошибкой.

**ENVBRK**

char \*envbrk(n)

unsigned n;

Работает идентично sbrk, за исключением того, что он всегда возвращает четное значение (указатель с нулевым младшим битом). Это достигается путем пропуска одного байта, если sbrk возвращает нечетное значение для данного аргумента n. В случае успеха envbrk возвращает указатель на первую ячейку памяти в добавленном блоке, в противном случае возвращает значение ERROR (-1). Отказ может быть вызван: (1) перекрытием 2 сохраненных программ, (2) перекрытием стека времени выполнения или (3) превышением доступной памяти. Обратите внимание, что n беззнаковое.

Также см. функции sbrk, ubrk, wrdbrk и brk.

**EXEC**

RESULT exec(fspec)

FILESPEC fspec;

Выполняет межпрограммный переход к файлу, указанному в строке с завершающим нулем, на которую указывает fspec. Функция exec - это вызов execl с двумя аргументами: execl(fspec, 0). Она включена для совместимости с BDS C. См. execl ниже.

**EXECL**

RESULT execl(fspec, arg0, arg1, arg2,...argn)

FILESPEC fspec;

char \*arg0, \*arg1, \*arg2,...\*argn;

Выполняет межпрограммный переход к файлу, указанному в строке с завершающим нулем, на которую указывает fspec. То есть он загружает и выполняет код, который, как предполагается, содержит файл fspec. Межпрограммное переключение иногда называют цепочкой программ. Функция execl позволяет вам последовательно выполнять серию программ, при этом каждая программа в этой серии накладывается на образ памяти предыдущей программы.

Параметры командной строки могут быть переданы в вызываемую программу в виде последовательности строк с завершающим нулем, на которые указывают аргументы arg1, arg2, ... argn. Последний аргумент (argn) должен быть нулевым. Это сделано для совместимости с UNIX. arg0 игнорируется в CP/M из-за неудачной особенности этой операционной системы. Функция execl создает командную строку из строк, на которые указывают ее аргументы (в CP/M и MS-DOS это означает чередование пробелов между аргументами).

Данные также могут передаваться от вызывающей программы к вызываемой программе в файлах или через область внешних данных. Чтобы передать данные внутри файла, вызывающая программа должна закрыть файл, а вызываемая программа должна снова открыть его. Чтобы передавать данные (включая дескрипторы открытых файлов) через область внешних данных, источник этой области должен быть одинаковым для обеих программ. В CP/M-86 и MS-DCS этого можно было добиться, задав в обеих программах один и тот же набор глобальных объектов.

Функция execl возвращает ERROR (-1), поскольку любой возврат из execl является ошибкой. Это произойдет, если спецификация файла не существует или не может быть прочитана по какой-либо причине, что предотвращает переполнение образа памяти вызывающей программы и выполнение вызванной программы. Вызванная программа может вернуться к вызывающей программе через другой вызов execl.

execl - это функция list. (См. [*Главу 2, Использование функций list*](#_Использование_функций_list).)

**EXIT**

exit(i)

int i;

Передает управление от программы обратно операционной системе (вызывает выход на системный уровень). выход никогда не возвращается. Он вызывает fclose, чтобы закрыть stdout и stderr (стандартный файл перенаправленного вывода и стандартный файл ошибок. См. xmain). Никакие другие файлы не сбрасываются (записываются на диск), ни один из открытых файлов не закрывается. (См. K&R, стр. 154.) Параметр i в настоящее время игнорируется в CP/M и MS-DOS. Если требуется выйти из операционной системы без каких-либо операций ввода-вывода, вызовите ccexit вместо exit.

**EXTERNS**

char \*externs()

Возвращает указатель на первый байт в области внешних данных вашей программы. Если вы не повторно создали эту область, это значение будет таким же, как и возвращаемое codend. (В настоящее время не реализована.)

**FABORT**

RESULT fabort(fd)

FILE \*fd;

Освобождает файловый дескриптор, выделенный для открытого файла, не закрывая этот файл. Файл определяется указателем fd на его файловый дескриптор. Вызов fabort не повлияет на содержимое файла, открытого только для ввода, но вызов его для файла, открытого для вывода, может привести к потере некоторых или всех данных, записанных в этот файл. (Мы включили эту функцию для совместимости с BDS C - мы не рекомендуем ее использовать.)

Функция fabort, в случае успеха, возвращает SUCCESS (0). Функция fabort возвращает ERROR (-1), если fd не указывает на допустимый файловый дескриптор.

**FCLOSE**

fclose(fd)

STREAM fd;

Закрывает файл, открытый для буферизованного вывода через fopen. Файл определяется указателем fd на его файловый дескриптор. Функция fclose помещает символ конца файла (EOF) в текущую позицию в буфере ввода-вывода файла и вызывает fflush перед закрытием файла.

Функция fclose возвращает SUCCESS (0), если указанный файл был успешно закрыт. Функция fclose возвращает ERROR (-1) и не закрывает файл, если: (1) указанный файл не был открыт для буферизованного вывода через fopen, (2) fd не указывает на действительный дескриптор файла или (3) файл не удалось закрыть из-за ошибки на уровне операционной системы (см. K&R, стр. 153).

**FDOPEN**

STREAM fdopen(fd, mode, buffer\_size)

FILE \*fd;

char \*mode;

unsigned buffer\_size;

Преобразует небуферизованный файловый дескриптор (FILE \*) в STREAM (буферизованный файловый дескриптор). режим mode должен быть совместим с атрибутами чтения/записи fd. Параметр buffer\_size - это размер буфера, который будет использоваться дескриптором STREAM. Обратите внимание, что его можно использовать для изменения размера дескриптора STREAM, если он используется вместе с fileno, как в:

**strn = fdopen(fileno(stm),mode,buffer\_size);**

В случае неудачи возвращает NULL.

**FERROR**

int ferror(fd)

STREAM fd;

Возвращает значение ошибки для данного потока. Значение ошибки, считываемое из errno, устанавливается при ошибке. Чтобы очистить его, используйте clearerr (описано ранее). В настоящее время существует только одно значение ошибки.

**FFLUSH**

RESULT fflush(fd)

STREAM fd;

Сбрасывает или записывает в файл текущее содержимое буфера ввода-вывода, связанного с файлом, открытым для буферизованного вывода через fopen. Файл определяется указателем fd на его файловый дескриптор. Размер буфера ввода-вывода устанавливается при открытии файла. После вызова fflush указатель ввода-вывода файла будет указывать сразу за последним использованным байтом, как и ожидалось.

Функция fflush возвращает SUCCESS (0), если буфер был успешно записан в файл. (Вызов fflush, когда буфер пуст, не имеет никакого эффекта, кроме возврата SUCCESS.) Функция fflush возвращает ERROR (-1) и не очищает буфер, если: (1) файл не был открыт для буферизованного вывода через fopen, ( 2) fd не указывает на допустимый файловый дескриптор или (3) все содержимое буфера не может быть записано из-за ошибки на уровне операционной системы (см. K&R, стр. 166).

**FGETC**

int fgetc(fd)

STREAM fd;

Идентичен getc (описан позже). Гарантированно будет функцией, а не макросом препроцессора.

**FGETS**

char \*fgets(s, n, fd)

char \*s;

unsigned n;

STREAM fd;

Считывает не более n-1 символов (байтов) из файла, открытого для буферизованного ввода через команду fopen, в строку, начинающуюся с s. Файл определяется указателем fd на его файловый дескриптор. Функция fgets не будет читать после символа новой строки или более n-1 символов, в зависимости от того, что наступит раньше. Затем fgets добавляет нулевой символ к считанным символам для создания строки с завершающим нулем.

Функция fgets в случае успеха возвращает указатель на эту строку (идентичный значению, переданному в s). Функция fgets возвращает значение нулевого указателя (0), если: (1) fd не указывает на действительный дескриптор файла, (2) файл не может быть прочитан из-за ошибки на уровне операционной системы или (3) конец файл был достигнут (см. K&R, стр. 155).

**FILENO**

FILE \*fileno(sfd)

STREAM sfd;

Возвращает дескриптор файла, связанный с потоком sfd.

**FOPEN**

STREAM fopen(fspec, mode, buffer\_size)

FILESPEC fspec;

char \*mode;

int buffer\_size;

Создает и/или открывает файл для буферизованного ввода-вывода со спецификацией файла, указанной в fspec. Он использует open (см. Функцию open далее в этой главе) для фактического открытия файла. Таким образом, можно использовать полные спецификации файла. Если fopen успешна, она возвращает указатель на действительный файловый дескриптор для указанного файла. Вы должны сохранить этот указатель для последующего использования в вашей программе. Функция fopen возвращает NULL (0) и не создает и не открывает какой-либо файл, если: (1) файловый режим не распознан, (2) недостаточно пользовательской памяти для нового дескриптора файла, (3) указанная спецификация файла недействительна или (4) файл не может быть открыт или создан из-за ошибки на уровне операционной системы.

Для режима (mode) вы должны указать одно из следующего: "w", "a" или "r". То, что вы укажете, определяет режим ввода-вывода файла, как указано в следующей таблице:

|  |  |
| --- | --- |
| "w" | режим только для записи |
| "a" | режим только для записи, добавить вывод в конец файла |
| "r" | режим только для чтения |

Если режим (mode) ввода-вывода файла - "w" или "a", говорят, что он открыт для буферизованного вывода. Если режим ввода-вывода файла - "r", говорят, что он открыт для буферизованного ввода.

Значение, которое вы указываете для buffer\_size, определяет размер буфера ввода-вывода, связанного с файлом. Это значение лучше всего установить как положительное целое кратное от размера системной записи. (Системная запись - это минимальная единица данных, передаваемых во время операций файлового ввода-вывода. В CP/M размер системной записи составляет 128 байт. В UNIX это 512 байт. Для получения дополнительной информации обратитесь к документации вашей операционной системы.) Указатель до первого байта в этом буфере ввода-вывода сохраняется в файловом дескрипторе. Таким образом, любой другой функции буферизованного файлового ввода-вывода необходимо передать только указатель на этот файловый дескриптор (см. K&R, стр. 151, 167).

**FPRINTF**

RESULT fprintf(fd, format, arg1, arg2, ...)

FILE \*fd;

char \*format;

TYPE arg1;

TYPE arg2;

...

Идентичен printf, за исключением того, что вместо записи в стандартный вывод он записывает свою отформатированную строку вывода в буфер ввода-вывода, связанный с файлом, открытым для буферизованного вывода через fopen, начиная с текущего места в этом буфере. Когда этот буфер заполняется, он автоматически очищается (т.е. все его содержимое записывается в файл). Файл определяется идентификатором указателя на его файловый дескриптор.

fprintf - это функция list. См. [*Стр. 30*](#_Использование_функций_list_1).

Функция fprintf возвращает SUCCESS (0), если вся ее выходная строка была успешно записана в буфер. Однако из-за буферизации операций файлового ввода-вывода такое возвращаемое значение не может гарантировать, что эта же строка будет успешно записана в файл, поскольку ошибки, возникающие в результате и влияющие на результат конкретного вызова fprintf, могут не проявиться до тех пор, пока некоторый последующий вызов функции приводит к сбросу буфера ввода-вывода этого файла. Функция fprintf возвращает ERROR (-1), если: (1) fd не указывает на действительный дескриптор файла, (2) файл не был открыт для буферизованного вывода через fopen, (3) файл не может быть записан из-за ошибки на уровне операционной системы, или (4) невозможно записать всю строку в файл из-за нехватки дискового пространства (см. K&R, стр. 152).

**FPUTC**

RESULT fputc(c, fd)

char c;

STREAM fd;

Записывает символ c в STREAM fd. Идентичен putc (описанному позже), за исключением того, что он гарантированно не является макросом препроцессора.

**FPUTS**

RESULT fputs(c, fd)

char \*s;

STREAM fd;

Записывает строку с завершающим нулем, на которую указывает s, в буфер ввода-вывода, связанный с файлом, открытым для буферизованного вывода через команду fopen, начиная с текущего места в этом буфере. Когда этот буфер заполняется, он автоматически очищается (т.е. все его содержимое записывается в файл). Файл определяется указателем fd на его файловый дескриптор. Для каждого символа новой строки ('\ n'), появляющегося в строке, в буфер записываются возврат каретки и новая строка ("\ r \n" ). Терминальный нулевой символ не записывается.

Функция fputs возвращает ERROR (-1) и не записывает строку, если: (1) fd не указывает на допустимый файловый дескриптор, (2) файл не был открыт для буферизованного вывода через fopen или (3) файл не мог быть записанным из-за ошибки на уровне операционной системы. В противном случае fputs возвращает количество байтов, фактически записанных в буфер, минус количество вставленных символов возврата каретки. Однако из-за буферизации операций файлового ввода-вывода такое возвращаемое значение не гарантирует, что те же самые байты будут успешно записаны в файл, поскольку ошибки, возникающие в результате и влияющие на результат конкретного вызова fputs, могут не стать очевиден до тех пор, пока некий более поздний вызов функции не приведет к сбросу буфера ввода-вывода этого файла (см. K&R, стр. 155).

**FREAD**

int fread(buf, fsizeofitem, nitems, fd)

TYPE \*buf;

unsigned fsizeofitem;

unsigned nitems;

STREAM fd;

Считывает количество элементов (nitems) из STREAM fd. Размер отдельного элемента - fsizeofitem. Параметр buf - это адрес для чтения. Функция fread возвращает количество прочитанных элементов или ERROR (-1) при ошибке. Функцию fread можно перемежать с getc, fgetc, get2b и getw (все они описаны в этой главе). Выравнивание не имеет значения. См. также read, описанную ниже.

**FREE**

free(p)

char \*p;

Освобождает блок в памяти, ранее выделенный вызовом alloc. Аргумент p, который должен быть идентичен значению, возвращаемому этим вызовом alloc, является указателем на первую ячейку памяти в блоке. Выделенные слоты можно освобождать в любом порядке. Вызов free с аргументом, ранее не полученным вызовом alloc, является серьезной ошибкой. (см. K&R, стр.97, 177)

**FREOPEN**

int freopen(fspec, mode, buf\_size, fs)

FILE \*fspec;

STREAM fs;

char \*mode;

unsigned buf\_size;

Перенаправляет fd, как если бы был вызван fs=fopen(fspec, mode, buf\_size). В случае неудачи возвращает NULL.

**FSCANF**

RESULT fscanf(fd, format, arg1, arg2, ...)

FILE \*fd;

char \*format, arg1, arg2, ...;

TYPE \*arg1;

TYPE \*arg2;

...

Идентична scanf, за исключением того, что входная строка читается из буфера ввода-вывода, связанного с файлом, открытым для буферизованного ввода через команду fopen, а не из стандартного ввода. Файл определяется указателем fd на его файловый дескриптор, fscanf начинает чтение с текущей позиции в буфере ввода-вывода. Она прекращает чтение, когда успешно присвоила значения байтам, соответствующим каждому элементу, указанному в строке формата, или когда она достиг конца файла - в зависимости от того, что наступит раньше.

fscanf - это функция list, см. [*Использование функций list в конце главы 2*](#_Использование_функций_list_2).

Если ошибок нет, fscanf возвращает количество успешно присвоенных значений. fscanf возвращает ERROR (-1) и не выполняет ввода, если: (1) fd не указывает на допустимый файловый дескриптор, (2) файл не был открыт для буферизации через fopen или (3) файл не может быть прочитан из-за на уровне операционной системы (см. K&R, стр. 152).

**FWRITE**

int fwrite(buf, fsizeofitem, nitems, fd)

TYPE \*buf;

unsigned fsizeofitem;

unsigned nitems;

STREAM fd;

Записывает количество элементов (nitems) в STREAM fd. Размер отдельного предмета - fsizeofitem. Параметр buf - это адрес для записи. функция fwrite возвращает количество записанных элементов или ERROR (-1) в случае ошибки. Вызовы fwrite можно перемежать с putc, fputc, put2b и putw (все они описаны в этой главе). Выравнивание не имеет значения. Также см. write описанную ниже.

**GETC**

int getc(fd)

STREAM fd;

Возвращает один символ (байт) в виде целого числа (от 0 до 255 включительно) из файла, открытого для буферизованного ввода через fopen. Файл определяется указателем fd на его файловый дескриптор. Возврат каретки и перевод строки возвращаются явно. getc возвращает ERROR (-1), если: (1) файл не был открыт для буферизованного ввода или (2) был достигнут конец файла (см. K&R, стр. 152, 166).

**GETCHAR**

char getchar()

Возвращает следующий символ из стандартного ввода (устройство CP/M или MS-DOS CON: - обычно с клавиатуры консоли). Если getchar встречает Ctrl-Z (маркер EOF CP/M), он возвращает ошибку (-1). (См. K&R, стр. 13, 40, 144, 152, 162.)

**GETS**

gets(s)

char \*s

Считывает следующую строку из стандартного ввода (устройство CP/M или MS-DOS CON: - обычно с клавиатуры консоли) в строку, начинающуюся с s. gets заменяет символ новой строки ('\r') или комбинацию возврата каретки/новой строки ("\r\n"), которая завершает строку ввода нулевым символом для создания строки с нулевым символом в конце. Поскольку gets не проверяет, является ли строка, начинающаяся с s, достаточно длинной, чтобы содержать строку ввода, вы должны определить эту строку так, чтобы она могла содержать самую длинную строку ввода, которую вы могли разумно ожидать.

**GETW**

int getw(fd)

STREAM fd;

Последовательно возвращает одно целое число из файла, открытого для буферизованного ввода через fopen. Файл определяется указателем fd на его файловый дескриптор. Возврат каретки и перевод строки возвращаются явно, getw возвращает ERROR (-1), если: (1) файл не был открыт для буферизованного ввода, или (2) достигнут конец файла, или (3) если целое число равно ERROR (-1) появляется во входном файле. Таким образом, errno следует проверить на наличие истинных ошибок. Вызовы getc могут чередоваться с вызовами getw. Информация, прочитанная getc и getw, может быть записана с помощью putc и putw. Во входном файле не требуется особого выравнивания информации.

**GET2B**

int get2b(fd)

STREAM fd;

Возвращает двухбайтовое количество из буферизованного входного потока. Функция get2b похожа на getw и fread, за исключением того, что она инвариантна относительно порядка байтов.

**INDEX**

char \*index(s, c)

char \*s, c;

Возвращает указатель на первое вхождение символа c в строке, начинающейся с s. Функция index возвращает значение нулевого указателя (0), если c не встречается в строке. См. функцию rindex, описанную далее в этой главе.

**INITB**

initb(array, s)

char \*array, \*s;

Обеспечивает относительно удобную инициализацию символьных массивов. Следует передать два параметра: первый, array, должен быть указателем на массив символов. Второй, s, должен быть указателем на строку с завершающим нулем, содержащую символы ASCII, представляющие десятичное целое значение, разделенные запятыми. При вызове initb последовательно преобразует каждое десятичное целое число в строке, начинающейся с s, в двоичное целое число и присваивает 8 младших битов этого значения соответствующему элементу в массиве символов, на который указывает array.

Если в строке имеется n целых значений, а в массиве больше n элементов, значения будут присвоены только первым n элементам массива, а содержимое оставшихся элементов останется неизменным. Если в строке n целочисленных значений и меньше n элементов в массиве, байтам за пределами массива будут присвоены значения, как если бы они были элементами массива, и данные могут быть перезаписаны по ошибке. Программист несет ответственность за предотвращение или обеспечение таких ситуаций.

**INITW**

initb(array, s)

int \*array;

char \*s;

Обеспечивает относительно удобную инициализацию целочисленных массивов. Ему следует передать два параметра: первый, array, должен быть указателем на массив целых чисел. Второй, s, должен быть указателем на завершающуюся нулем строку символов ASCII, представляющую десятичные целые значения, разделенные запятыми. При вызове initb преобразует каждое десятичное целочисленное значение в строке, начинающейся с s, последовательно, до двоичного целочисленного значения и присваивает это значение соответствующему элементу в целочисленном массиве, на который указывает array.

Если в строке имеется n целых значений, а в массиве больше n элементов, только первые n элементов массива будут иметь значения, а содержимое остальных элементов останется неизменным. Если в строке n целочисленных значений и меньше n элементов в массиве, байтам за пределами массива будут присвоены значения, как если бы они были элементами массива, и данные могут быть перезаписаны по ошибке. Программист несет ответственность за предотвращение или обеспечение таких ситуаций.

**INP**

char inp(port)

int port;

Возвращает байтовое значение, присутствующее в указанном входном порту, после выполнения машинной инструкции byte IN для этого порта. (Эта функция доступна только на машинах, для которых имеет смысл инструкция byte IN или ее эквивалент.)

**ISASCII**

BOOL isascii(c)

char c;

Возвращает TRUE (1), если c является символом ASCII, в противном случае возвращает FALSE (0).

**ISATTY**

BOOL isatty(fd)

FILE \*fd;

Возвращает TRUE (1), если этот файловый дескриптор является терминалом; в противном случае - FALSE (0). Возвращает TRUE, если описатель файла ссылается на консоль ("CON:").

**ISCNTRL**

BOOL iscntrl(c)

char c;

Возвращает TRUE (1), если c - управляющий символ ASCII, в противном случае возвращает FALSE (0).

**INP16**

char inp16(port)

int port;

Функция **inp16**, недоступная в CP/M-80, возвращает значение, присутствующее в указанном входном порте, после выполнения 16-битной машинной инструкции IN для этого порта. (Эта функция доступна только на машинах, для которых имеет смысл 16-битная инструкция IN или ее эквивалент.)

**ISALNUM**

BOOL isalnum(c)

char c;

Возвращает TRUE (1), если c - буквенно-цифровой символ ASCII, в противном случае возвращается FALSE (0).

**ISALPHA**

BOOL isalpha(c)

char c;

Возвращает TRUE (1), если c - это алфавитный символ ASCII, иначе она возвращает FALSE (0) (см. K&R, стр. 127, 156).

**ISDIGIT**

BOOL isdigit(c)

char c;

Возвращает TRUE (1), если c является символом ASCII, представляющим одну из десятичных цифр 0-9, в противном случае возвращает FALSE (0) (см. K&R, стр. 127, 156).

**ISFD**

BOOL isfd(fd)

FILE \*fd;

Возвращает TRUE, если fd - допустимый файловый дескриптор, в противном случае - FALSE.

**ISHEAP**

BOOL isheap(ptr)

char \*ptr;

Возвращает TRUE, если ptr указывает на область данных, возвращенную из malloc. Должна вызываться только после первого вызова malloc.

**ISLOWER**

BOOL islower(c)

char c;

Возвращает TRUE (1), если c является буквенным символом нижнего регистра ASCII, в противном случае возвращает FALSE (0). (см. K&R, стр. 156).

**ISNUMERIC**

BOOL isnumeric(c, radix)

char c;

int radix;

Возвращает TRUE (1), если c - символ ASCII, представляющий действительную цифру в системе счисления с основанием, указанным в radix, в противном случае возвращается FALSE (0). Например,

isnumeric('A',15)

возвращает TRUE. Функция isnumeric определена только если 1 < radix < 36.

**ISPRINT**

BOOL isprint(c)

char c;

Возвращает TRUE (1), если c - печатаемый символ ASCII; в противном случае возвращается FALSE (0).

**ISPUNCT**

BOOL ispunct(c)

char c;

Возвращает TRUE (1), если c - символ ASCII, представляющий знак препинания, в противном случае возвращается FALSE (0).

**ISSPACE**

BOOL isspace(c)

char c;

Возвращает TRUE (1), если c - символ ASCII, представляющий пробел, табуляцию или новую строку. В противном случае возвращается FALSE (0). (Эта функция включена для совместимости с BDS C. Стандартная функция UNIX C - iswhite. Однако см. K&R, стр. 156.)

**ISUPPER**

BOOL isupper(c)

char c;

Возвращает TRUE (1), если c является алфавитным символом верхнего регистра ASCII, в противном случае возвращает FALSE (0) (см. K&R, стр. 145, 156).

**ISWHITE**

BOOL iswhite(c)

char c;

Возвращает TRUE (1), если c является символом ASCII, представляющим пробел, табуляцию, новую строку или перевод страницы, в противном случае возвращает FALSE (0).

**KBHIT**

BOOL kbhit()

Проверяет, был ли набран символ на клавиатуре консоли - возвращает TRUE, если это так, и FALSE, если нет. Точнее, kbhit возвращает TRUE (ненулевое значение), если символ присутствует на стандартном вводе (устройство CP/M или MS-DOS CON: обычно это клавиатура консоли). В противном случае возвращается FALSE (0). Эта функция недоступна в системах, которые не имеют такой функции (например, UNIX).

**LINK**

link(ofspec, nfsspec)

FILESPEC ofspec;

FILESPEC nfspec;

Делает nfsspec синонимом имени файла ofspec. Невозможно в CP/M или MS-DOS. В этих системах ссылка работает так же, как rename (описывается позже).

**LOOK**

look()

Блокирует процесс в быстрой памяти. Не работает в CP/M и MS-DOS.

**LONG арифметика**

Файл LONG.C содержит машинно-независимый пакет целочисленных арифметических операций LONG. Эти функции выполняют арифметические операции с 32-разрядными целыми числами с дополнением до двух. Все переменные, используемые этим пакетом, должны быть объявлены LONG или UNSIGNED LONG. Функции передают адрес каждого аргумента. Обычно функции не возвращают значения.

См. [*Приложение H*](#_Приложение_H_Функции_1) для описания функций длинных целых чисел.

**LONGJMP**

longjmp(savearea, i)

int savearea[];

int i;

Восстанавливает состояние программы из области сохранения. (savearea должна быть длиной 6 байтов для серий 8080 и 8086). Состояние программы включает в себя все регистровые переменные, счетчик программы возврата и указатель стека. Параметр savearea должен был ранее использоваться в качестве аргумента для setjmp (описан позже). После вызова longjmp с той же областью сохранения состояние восстанавливается, фактически выглядя так, как если бы произошел возврат из setjmp, с возвращаемым значением i. Функция longjmp - это обобщенная версия сброса.

**MALLOC**

char \*malloc(n)

unsigned n;

Выделяет непрерывную область памяти длиной n. Каждый выделяемый блок начинается с четного адреса.

Функция malloc в случае успеха возвращает указатель на первую ячейку памяти в блоке. Вы должны сохранить этот указатель для последующего использования в вашей программе. Функция malloc возвращает значение указателя NULL (0) и не выделяет никакой памяти, если выделение непрерывного блока запрошенного размера будет перекрывать: (1) сохраненную программу, (2) стек времени выполнения или (3) предыдущий выделенный блок в памяти. (См. Описание alloc от K&R на стр. 97, 175)

**MAX**

int max(a, b)

int a, b;

Возвращает большее из a или b.

**MIN**

int min(a, b)

int a, b;

Возвращает меньшее из a или b.

**MKTEMP**

mktemp(f)

FILESPEC f;

Изменяет строку FILESPEC на уникальное имя файла: возвращаемая строка может использоваться в fopen, open или creat без конфликта с существующим файлом. Если mktemp не может сформировать уникальное имя файла, она возвращает NULL. Если это возможно, она возвращает свой аргумент f, mktemp ищет заглавную X и заменяет ее и следующий за ней символ двузначным десятичным числом.

**MOVMEM**

movmem(source, dest, n)

char \*source, \*dest;

int n;

Копирует содержимое n смежных байтов в памяти, начиная с source, в n смежных байтов, начинающихся с dest. Нет ограничений на перекрытие этих двух областей. Байты в области, на которую указывает источник, не изменяются, если они не перезаписываются в результате совпадения областей.

**NICE**

nice(n)

int n;

Устанавливает приоритет текущего процесса. Нулевая процедура в CP M и MS-DOS.

**OPEN**

FILE \*open(fspec, mode)

FILESPEC fspec;

unsigned mode;

Открывает файл, указанный в fspec, для прямого ввода-вывода. Этот файл уже должен существовать. (См. creat.) Функция open, в случае успеха, возвращает указатель на действительный файловый дескриптор для указанного файла. Вы должны сохранить этот указатель для последующего использования в вашей программе. Функция open возвращает ERROR (-1) и не открывает файл, если: (1) недостаточно памяти пользователя для нового файлового дескриптора, (2) Данная спецификация файла недействительна, (3) указанный файл либо не существует, либо не был создан с помощью creat, или (4) файл не может быть открыт из-за ошибки на уровне операционной системы.

Для mode вы должны указать одно из следующих значений: 0, 1 или 2. От того, какой из них вы укажете, зависит режим ввода-вывода файла, как указано в следующей таблице:

|  |  |
| --- | --- |
| 0 | режим только для чтения |
| 1 | режим только для записи |
| 2 | режим для чтения и записи |

(См. K&R, стр. 162.)

**OTELL**

unsigned int otell(fd)

FILE \*fd;

Возвращает байтовое смещение от начала текущего 512-байтового блока файла, с которого начнется следующая операция ввода-вывода файла для этого файла. Файл определяется указателем fd на его файловый дескриптор. Функция otell не указывает, в каком 512-байтовом блоке начнется операция ввода-вывода. Смотри, rtell и tell.

**OUTP**

outp(ptr, b)

int ptr;

int b;

Помещает b в выходной порт, указанный в ptr, и выполняет машинную инструкцию OUT byte для этого порта. (Эта функция доступна только на машинах, для которых инструкция OUT byte имеет смысл.)

**OUTP16**

outp16(ptr, w)

int ptr, w;

Помещает слово w в выходной порт, указанный в ptr, и выполняет машинную инструкцию OUT word для этого порта. (Эта функция доступна только на машинах, для которых OUT word имеет смысл.)

**PERROR**

perror(s)

char \*s;

Печатает строку s, затем двоеточие, а затем выводит строку, интерпретирующую значение errno, значение ошибки ввода-вывода. Если errno равно нулю, интерпретация не печатается.

**PGETC**

pgetc(fd)

STREAM fd;

Идентична getc, за исключением того, что она заменяет системный индикатор конца строки на '\n'. В CP/M и MS-DOS это означает, что всякий раз, когда он встречает символ возврата каретки ('\r'), за которым следует символ новой строки ('\n'), она возвращает только новую строку. Таким образом, pgetc преобразует строки в файлах из формата CP/M в формат UNIX (ANSI).

**POKE**

poke(addr, b)

char \*addr, b;

Записывает байт b в байт памяти по адресу addr. b должно быть выражением lvalue (см. K&R, стр. 183). Функция poke, которая была включена для совместимости с BDS C, является избыточной в C, поскольку косвенное обращение является особенностью языка.

**PPUTC**

pputc(c, fd)

char c;

STREAM fd;

Идентична putc, за исключением того, что она связывает символ '\n' с системным символом конца строки. В CP/M и MS-DOS это означает, что всякий раз, когда он передает символ новой строки ('\n'), она сначала записывает символ возврата каретки ('\r') в буфер ввода-вывода файла, а затем записывает переданный символ новой строки. Таким образом, pputc преобразует строки, записанные в файлы из UNIX (ANSI), в формат CP/M.

**PRINTF**

printf(format, arg1, arg2, ...)

char \*format;

TYPE \*arg1;

TYPE \*arg2;

...

Записывает форматированную строку вывода в стандартный вывод (устройство CP/M или MS-DOS CON: обычно экран консоли. Функция printf должна передавать указатель, формат, в строку с завершающим нулем (строковая константа также является допустимо для формата, так как оно оценивается как строка с завершающим нулем. Эта строка управляет генерацией выходной строки, в printf может быть передано несколько. других аргументов: arg1, arg2, .... Отдельные аргументы в этой серии могут быть символами, целыми числами, целыми числами без знака или строковыми указателями. Обязателен только первый аргумент, format, все остальные необязательны.

printf - это функция list. См. [*Использование функций list*](#_Использование_функций_list_3) в конце главы 2.

Строка, на которую указывает format, может содержать как обычные символы, так и специальные подстроки, начинающиеся с символа %, которые называются спецификациями преобразования. Каждый обычный символ, обнаруженный printf при сканировании строки справа налево, просто записывается в стандартный вывод. Каждая спецификация преобразования, если встречается, вызывает преобразование и форматирование значения следующего аргумента в серии arg1, arg2, ..., как указано, и запись в стандартный вывод.

После символа % в каждой спецификации преобразования может стоять:

1. необязательный знак минус '-', который, если он присутствует, заставляет преобразованное значение корректировать влево в его поле. Регулировка вправо по умолчанию.
2. необязательная строка десятичных цифр, определяющая минимальное количество символов в поле, в которое должно быть записано значение. Преобразованное значение никогда не будет усечено. Однако, если в нем меньше символов, чем указано здесь, оно будет дополнено слева (или справа, если была указана настройка влево) пробелами до указанной ширины. Если эта строка цифр начинается с нуля, преобразованное значение будет дополнено нулями вместо пробелов.
3. другая необязательная строка десятичных цифр, которой должна предшествовать точка, '.', указывающая максимальное количество символов, которые должны быть скопированы из строки с завершающим нулем.
4. символ, называемый символом преобразования, указывающий тип преобразования, которое должно быть выполнено.

Из вышеперечисленного в спецификации преобразования должен присутствовать только символ преобразования. Все остальные, если они есть, должны быть в том порядке, в котором они перечислены.

Допустимые символы преобразования и указанные ими типы преобразований:

**c** младший байт аргумента интерпретируется как символ. Этот символ пишется только в том случае, если он пригоден для печати.

**d** Аргумент, который должен быть целым числом, преобразуется в десятичную систему счисления.

**o** аргумент, который должен быть целым числом, преобразуется в восьмеричную систему счисления.

**x** аргумент, который должен быть целым числом, преобразуется в шестнадцатеричное представление.

**u** аргумент, который должен быть целым числом без знака, преобразуется в десятичное представление.

**s** аргумент интерпретируется как указатель на строку. Символы из указанной строки читаются и записываются до тех пор, пока не будет прочитан нулевой символ или не будет записано необязательно указанное максимальное количество символов. См. пункт 3 выше.

**%** пишется символ %. Это escape-последовательность. Здесь нет никаких аргументов.

(См. K&R, стр. 7, 11, 145.)

**PUTC**

putc(c, fd)

char c;

STREAM fd;

Записывает символ c в буфер ввода-вывода, связанный с файлом, открытым для буферизованного вывода через fopen, начиная с текущего места в этом буфере. Когда этот буфер заполняется, он автоматически очищается (то есть все его содержимое записывается в файл). Файл определяется указателем fd на его файловый дескриптор.

Функция putc возвращает ERROR (-1) и не записывает символ, если: (1) fd не указывает на допустимый дескриптор файла, (2) файл не был открыт для буферизованного вывода через fopen или (3) буфер не удалось записать из-за ошибки на уровне операционной системы. В противном случае putc записывает символ в буфер ввода-вывода файла и возвращает SUCCESS (0). Однако из-за буферизации операций ввода-вывода файла такое возвращаемое значение не гарантирует, что тот же самый символ будет успешно записан в файл, поскольку ошибки, возникающие в результате и влияющие на результат конкретного вызова putc, могут не стать очевидными. пока какой-либо более поздний вызов функции не приведет к сбросу буфера ввода-вывода этого файла (см. K&R, стр. 152, 166).

**PUTCHAR**

putchar(c)

char c;

Записывает символ c в стандартный вывод (устройство CP/M или MS-DOS CON: - обычно экран консоли). (См. K&R, стр. 13, 144, 152).

**PUTDEC**

putdec(nn)

int nn;

Печатает десятичное число nn на консоли. См. printf для получения более подробной информации.

**PUTS**

puts(s)

char \*s;

Записывает строку, начинающуюся с s, в стандартный вывод (устройство CP/M или MS-DOS CON: - обычно экран консоли). Все команды каретки должны явно указываться в этой строке.

**PUTW**

putw(i, fd)

int i;

STREAM fd;

Записывает целое число i в буфер ввода-вывода, связанный с файлом, открытым для буферизованного вывода через команду fopen, начиная с текущего местоположения в этом буфере. Когда этот буфер заполняется, он автоматически очищается (то есть все его содержимое записывается в файл). Файл определяется указателем fd на его файловый дескриптор.

Функция putw возвращает ERROR (-1) и не записывает целое число, если: (1) fd не указывает на действительный дескриптор файла, (2) файл не был открыт для буферизованного вывода через fopen или (3) буфер не удалось записать из-за ошибки на уровне операционной системы. В противном случае putw записывает целое число в буфер ввода-вывода файла и возвращает SUCCESS (0). Однако из-за буферизации операций ввода-вывода файла такое возвращаемое значение не гарантирует, что то же самое целое число будет успешно записано в файл, поскольку ошибки, возникающие в результате и влияющие на результат конкретного вызова putw, могут не стать очевидными. пока какой-либо более поздний вызов функции не приведет к очистке буфера ввода-вывода файла. Вызовы putс и putw могут чередоваться. Файлы, написанные с помощью putc и putw, можно прочитать с помощью getc и getw.

**PUT2B**

put2b(i, fd)

int i;

STREAM fd;

Выводит двухбайтовое количество в буферизованный выходной поток. Функция put2b похожа на putw и fwrite, за исключением того, что он инвариантен относительно порядка байтов.

**QSORT**

int qsort(tbl, nregs, reclen, cmp)

char tbl[][];

unsigned nregs, reclen;

int (\*cmp)();

Функция qsort выполняет сортировку по возрастанию для двумерного массива

**tbl[nrecs] [reclen]**

То есть tbl указывает на основание массива; reclen - длина записи, подлежащей сортировке; nregs - количество записей; cmp - функция, используемая для выполнения сравнения. Она будет вызываться с двумя указателями на записи. По сути, ее декларация-это:

**int**

**cmp(a,b)**

**char a[reclen];**

**char b[reclen];**

**{**

Она должна возвращать значение меньше нуля, если сравнение "меньше чем". Нулевое значение, если сравнение "такое же", в противном случае она должна возвращать значение больше нуля.

**RAND**

int rand()

Возвращает следующее значение в последовательности псевдослучайных чисел, инициализированной предыдущим вызовом srand. Значения в последовательности будут находиться в диапазоне от 0 до 65535.

Выражение C

**rand() % n**

будет оцениваться как целое число, большее или равное 0, но меньше n.

**READ**

int read(fd, bufr, n)

FILE \*fd;

TYPE \*bufr;

unsigned n;

Считывает максимум n байтов из файла, открытого для прямого или буферизованного ввода, начиная с текущего местоположения указателя ввода-вывода файла, в буфер памяти, на который указывает bufr. Файл определяется указателем fd на его файловый дескриптор.

Вы должны определить буфер, на который указывает bufr, так, чтобы он мог содержать не менее n байтов.

Указатель файлового ввода-вывода всегда будет указывать на начало системной записи. После вызова чтения указатель ввода-вывода файла укажет на начало системной записи, следующей за последней прочитанной.

Если ошибок нет, read возвращает фактическое количество прочитанных байтов. Если эти байты считываются из файла, read возвращает либо кратное размеру системной записи, либо ноль (0). Ноль будет возвращен только в том случае, если будет достигнут конец файла. Если байты читаются с последовательного устройства (такого как устройства CP/M или MS-DOS CON: или RDR:), открытого как файл, чтение возвращает единицу (1), так как только один байт за вызов read может быть прочитан с последовательного устройства. Функция read возвращает ERROR (-1) и не пытается прочитать файл, если: (1) файл не был открыт для ввода, (2) n меньше, чем размер системной записи, или (3) файл не мог быть прочитанным из-за ошибки на уровне операционной системы (см. K&R, стр. 163)

**REALLOC**

TYPE \*realloc(p, nbytes)

TYPE \*p;

unsigned nbytes;

Изменяет размер выделенной области, на которую указывает p (p должен быть предварительно установлен вызовом malloc). Функция realloc сохраняет содержимое области, насколько это возможно, поскольку область может иметь новый размер. Функция realloc возвращает указатель на область нового размера.

**RENAME**

RESULT rename(fname, fspec)

FILESPEC fname;

FILESPEC fspec;

Переименовывает файл, указанный в fspec, давая ему имя, содержащееся в строке с завершающим нулем, на которую указывает fname. (Строковая константа, такая как "newname", также действительна для fname, поскольку она оценивает указатель на строку с завершающим нулем.) Имя диска и номер, если таковые имеются, не изменяются.

**RESET**

reset(n)

int n;

Заставляет выполнение программы возвращаться к точке, установленной предыдущим вызовом setexit. Этот перевод имеет вид возврата из setexit. Параметр n, переданный для сброса, отображается как значение, возвращаемое setexit.

reset и setexit вместе позволяют более простое и понятное кодирование повторяющихся выходов к общей точке - особенно, когда такие передачи требуют перебора количества уровней вызовов функций. Например: при написании интерактивного редактора вы можете вызвать setexit в верхней части цикла команд и проверить, было ли его кажущееся возвращаемое значение равным нулю (0). Каждое ненулевое значение может использоваться для обозначения различных условий ошибки. Можно напечатать номер ошибки и продолжить выполнение цикла команд. Вызовы для reset будут разбросаны в соответствующих местах по всему циклу. В каждом случае параметр, переданный для reset, будет указывать на наличие (ненулевое значение) или отсутствие (ноль) определенного состояния ошибки.

reset и setexit, хотя они напоминают функции в использовании и синтаксисе, реализуются как директивы препроцессора компилятора, а не как функции. Таким образом, вы не найдете их ни в одном из файлов библиотек стандартных функций.

**RINDEX**

char \*rindex(s, c)

char \*s, c;

Возвращает указатель на последнее вхождение символа c в строке, начинающейся с s. Функция rindex возвращает значение нулевого указателя (0), если c не встречается в строке. См. функцию index, описанную ранее в этой главе.

**RTELL**

unsigned int rtell(fd)

FILE \*fd;

Возвращает смещение в 512-байтовых блоках от начала файла 512-байтового файлового блока, в пределах которого начнется следующая операция ввода-вывода файла с этим файлом. Функция rtell не указывает смещение в этом блоке, с которого начнется операция ввода-вывода. Файл указывается указателем fd в его файловый дескриптор. Смотри, otell и tell.

**SBRK**

char \*sbrk(n)

int n;

Добавляет n байтов в пользовательскую память (увеличивает CCEDATA на n). Функция sbrk в случае успеха возвращает указатель на первый байт в добавленном блоке. Функция sbrk возвращает значение ERROR (-1) и не добавляет байтов в пользовательскую память, если блок указанного размера: (1) перекрывает сохраненную программу, (2) перекрывает стек времени выполнения или (3) превышает доступную память.

ВАЖНО: не вызывайте sbrk с отрицательным аргументом между вызовами alloc (см. K&R, стр.175)

**SCANF**

RESULT scanf(format, arg1, arg2, ...)

char \*format;

TYPE \*arg1;

TYPE \*arg2;

...

Считывает форматированную строку ввода со стандартного ввода (устройство CP/M или MS-DOS CON: обычно с клавиатуры консоли). Под контролем строки формата, на которую указывает его первый аргумент, format, scanf извлекает серию подстрок, известных как поля ввода, из своей строки ввода, преобразует значения, представленные в каждом из этих полей, известные как значения ввода, и назначает их последовательно преобразованные значения в объекты, на которые указывают его оставшиеся аргументы arg1, arg2, ...

scanf - это функция list. См. [*Использование функций list*](#_Использование_функций_list_3) в конце главы 2.

В качестве первого аргумента scanf должен быть передан указатель format на соответствующую строку формата с завершающим нулем. (Строковая константа также действительна для формата, поскольку она оценивает указатель на строку с завершающим нулем.) В scanf может быть передан ряд других аргументов, arg1, arg2, ..., все из которых должны быть указателями . Отдельные объекты, на которые указывает arg1, arg2, ... могут быть символами, символьными массивами или целыми числами.

Строка формата может содержать пробельные символы (то есть пробелы, табуляции и символы новой строки), обычные символы или специальные подстроки, начинающиеся с символа %, известные как спецификации преобразования. Первая спецификация преобразования в строке формата соответствует и определяет границы первого поля ввода во входной строке. Она также определяет тип преобразования, которое будет выполнено для входного значения, представленного в этом поле. Каждая последующая пара спецификаций преобразования и полей ввода имеет то же отношение.

После символа % в каждой спецификации преобразования могут появиться:

1. необязательный символ подавления присваивания, '\*', который, если присутствует, заставляет пропустить соответствующее поле ввода.
2. необязательная строка десятичных цифр, определяющая максимальное количество символов в соответствующем поле ввода.
3. символ, называемый символом преобразования, указывающий тип преобразования, которое должно быть выполнено для соответствующего входного значения.

Из вышеперечисленного в спецификации преобразования должен присутствовать только символ преобразования. Все остальные, если они есть, должны быть в том порядке, в котором они перечислены выше.

Допустимые символы преобразования и указанные ими типы преобразований:

**%** в этот момент во входной строке ожидается один символ %. Это escape-последовательность - присвоение не выполняется.

**c** входное значение интерпретируется как символ. Соответствующий аргумент должен быть символьным указателем. Обычный пропуск пробелов подавляется. Чтобы прочитать следующий символ без пробела, используйте %ls. Если также задана ширина поля, то соответствующий аргумент должен быть указателем на массив символов, и указанное количество символов будет прочитано.

**s** входное значение интерпретируется как символьная строка соответствующий аргумент должен быть указателем на массив символов, достаточно большой, чтобы содержать строку в дополнение к терминальному нулевому символу, добавленному scanf. Поле ввода завершается либо пробелом, либо новой строкой, либо после считывания максимального количества символов, в зависимости от того, что наступит раньше.

**[** входное значение интерпретируется как строка символов. Соответствующий аргумент должен быть указателем на массив символов, достаточно большой, чтобы содержать строку плюс конечный нулевой символ, добавленный scanf. Окончание поля ввода определяется следующим образом. За левой квадратной скобкой выше следует набор символов и правая скобка. Если первый символ в этом наборе не является циркумфлексом, '^', поле ввода заканчивается первым символом в наборе в квадратных скобках. Если первый символ - это циркумфлекс, поле ввода заканчивается первым символом в наборе в квадратных скобках (за исключением символа '^').

**d** входное значение интерпретируется как десятичное целое число и преобразуется в двоичное целое число. Соответствующий аргумент должен быть целочисленным указателем.

**o** Входное значение интерпретируется как восьмеричное целое число и преобразуется в двоичное целое число. Соответствующий аргумент должен быть целочисленным указателем.

**x** входное значение интерпретируется как шестнадцатеричное целое число и преобразуется в двоичное целое число. Соответствующий аргумент должен быть целочисленным указателем.

Центральной задачей scanf является определение границ полей ввода в строке ввода, которые содержат входные значения, которые необходимо преобразовать и присвоить. Чтобы найти эти подстроки, scanf просматривает символы во входной строке, сравнивая каждый из них с соответствующими символами в строке, на которую указывает format. Если символ во входной строке совпадает с соответствующим символом в строке формата, он отбрасывается и читается следующий символ во входной строке. Если соответствующие символы не совпадают, scanf немедленно возвращается. Объединяет любое количество пробелов во входной строку соответствует любому количеству пробелов в строке формата. Пробел в строке формата является необязательным (он игнорируется, в то время как во входной строке он может ограничивать поля ввода. Таким образом, соответствующие символы - это не просто те символы, которые имеют одинаковое количество байтов от начала соответствующих строк. Символ %, который вводит спецификацию преобразования, встречается в строке формата, соответствующий символ в строке ввода считается первым байтом поля ввода. Поле ввода расширяется до тех пор, пока во входной строке не встретится символ пробела или количество байтов, указанное для ширины поля, прочитано, в зависимости от того, что наступит раньше. Символы преобразования c и [ выше являются единственными исключениями из этого общего правила. Любой несоответствующий символ в поле ввода вызывает немедленное возвращение scanf.

scanf возвращает либо количество преобразованных входных значений, которые он назначил, либо, если на стандартном входе нет входных данных, константу EOF (см. K&R, стр. 147).

**SEEK**

int seek(fd, offset, origin)

FILE \*fd;

int offset;

int origin;

Устанавливает значение указателя файлового ввода-вывода, связанного с открытым файлом. Указатель ввода-вывода файла должен быть от 0 до 8 мегабайт. Файл определяется указателем fd на его файловый дескриптор и может быть открыт для прямого или буферизованного ввода-вывода. Функция seek в основном используется в сочетании с функциями tell и прямого файлового ввода-вывода, чтения и записи. Функция seek должна использоваться с большей осторожностью в сочетании с функциями буферизованного файлового ввода-вывода, чтобы предотвратить потерю данных.

Значение, присвоенное offset, имеет различную интерпретацию в зависимости от значения, присвоенного origin:

Если origin равно 0, то указатель ввода-вывода файла будет указывать на начало файла плюс байты offset.

Если origin равно 1, то указатель ввода-вывода файла будет указывать на его текущую позицию в файле плюс байты offset.

Если origin равно 2, то указатель ввода-вывода файла будет указывать на конец файла плюс байты offset.

Если origin равно 3, то указатель ввода-вывода файла будет указывать на начало файла плюс offset, умноженное на 512 байт.

Если origin равно 4, то указатель ввода-вывода файла будет указывать на его текущую позицию в файле плюс offset, умноженное на 512 байт.

Если origin равно 5, то указатель ввода-вывода файла будет указывать на конец файла плюс offset, умноженное на 512 байт.

(См. K&R, стр. 164.)

**SETEXIT**

int setexit()

Вызов setexit устанавливает свое местоположение как точку "сброса" - точку, в которую последующие вызовы сброса выполнения программы передачи; возвращается 0. Каждый последующий вызов сброса вызывает очевидный возврат из функции setexit. Функция setexit, похоже, возвращает значение параметра n, которое было передано в reset. См. reset и setjmp.

**SETJMP**

int setjmp(savearea)

int savearea[savesize];

Вызов setjmp сохраняет состояние программы в savearea и возвращает 0. savearea должна быть длиной 6 байт, как для серий 8080, так и для 8086.) Состояние программы включает все регистровые переменные, счетчик программы возврата и указатель стека. . После вызова longjmp (см. предыдущее описание) с таким же сохранением, состояние восстанавливается, фактически выглядя так, как если бы произошел возврат из setjmp, а возвращаемое значение было предоставлено longjmp. Функция setjmp - это обобщенная версия setexit.

**SETMEM**

setmem(p, n, b)

TYPE \*p;

unsigned n;

char b;

Устанавливает n непрерывных байтов, начинающихся с p, в значение, указанное в b. Вы можете использовать setmem для инициализации различных буферов и массивов.

**SLEEP**

sleep(n)

unsigned n;

Приостанавливает выполнение на n десятых секунды на процессоре Z80, работающем на частоте 4 МГц. Вы можете адаптировать эту функцию к другому процессору и/или тактовой частоте, изменив значение одной или двух констант, расположенных в коде функции.

**SPRINTF**

sprintf(s, format, arg1, arg2, ...)

char \*s, \*format;

TYPE arg1;

TYPE arg2;

...

Идентична printf за исключением того, что записывает свой форматированный вывод в строку, начинающуюся с s. Сравните ее с printf, которая записывает свой вывод в стандартный вывод, и fprintf, которая записывает свой вывод в файл. sprintf добавляет нулевой символ к форматированной выходной строке (см. K&R, стр. 153). sprintf - это функция list.

**SRAND**

srand(seed)

int seed;

Инициализирует возвращаемое значение rand значением, переданным в seed.

**SSCANF**

sscanf(s, format, arg1, arg2, ...)

char \*s, \*format;

TYPE arg1;

TYPE arg2;

...

Идентичен scanf (и fscanf), за исключением того, что его форматированная входная строка считывается из строки с завершающим нулем, начинающейся с s, а не из стандартного ввода. sscanf не читает конечный нулевой символ (см. K&R, стр. 153). sscanf - это функция list.

**STRCAT**

char \*strcat(s1, s2)

char \*s1, \*s2;

Добавляет копию строки, начинающейся с s2, в конец строки, начинающейся с s1, создавая одну строку с завершающим нулем. Обратите внимание, что результирующая строка начинается с s1 и содержит один конечный нулевой символ.

strcat возвращает указатель на результирующую строку, идентичный параметру s1, который был передан (см. K&R, стр. 44).

**STRCMP**

int strcmp(s1, s2)

char \*s1, \*s2;

сравнивает строку, начинающуюся с s1, со строкой, начинающейся с s2. Это сравнение похоже на сравнение по алфавиту, основанное на числовых значениях соответствующих символов в двух строках. Это сравнение заканчивается, когда встречается первый нулевой символ в любой строке.

strcmp возвращает положительное целое число, ноль (0) или отрицательное целое число в зависимости от того, является ли строка, начинающаяся с s1, соответственно, больше, равна или меньше, чем строка, начинающаяся с s2 (см. K&R, стр. 101).

**STRCPY**

strcpu(s1, s2)

char \*s1, \*s2;

Копирует строку, начинающуюся с s2, в строку, начинающуюся с s1, останавливаясь после того, как был вставлен нулевой символ. Если длина строки, начинающейся с s2, больше, чем длина строки, начинающейся с s1, данные в байтах, следующие за последним, могут быть перезаписаны по ошибке (см. K&R, стр. 100)

**STREQ**

int \*streq(s1, s2)

char \*s1, \*s2;

Сравнивает символы в строках, начинающихся с s1 и s2, где n - количество символов (исключая конечный ноль) в строке, начинающейся с s2. Функция streq возвращает n, если соответствующие символы в двух строках идентичны, в противном случае возвращается ноль (0). Подобен substr (описан позже), за исключением того, что возвращает конец строки вместо начала.

**STRLEN**

int strlen(s)

char \*s;

Возвращает количество символов (исключая конечный ноль) в строке, начинающейся с s (см. K&R, стр. 36, 95, 93).

**STRNCAT**

char \*strncat(s1, s2, n)

char \*s1, \*s2;

int n;

Идентичен strcat, за исключением того, что strncat добавляет не более n символов из строки, начинающейся с s2 (с усечением справа), до конца строки, начинающейся с s1.

**STRNCMP**

int strncmp(s, t, n)

char \*s, \*t;

unsigned n;

Сравнивает строки, на которые указывают s и t. Сравнение останавливается на первом '\0' (например, strcmp) или после того, как будут просканированы n символов, в зависимости от того, что наступит раньше.

**STRNCPY**

char \*strncpy(s1, s2, n)

char \*s1, \*s2;

int n;

Идентичен strcpy, за исключением того, что strncpy копирует ровно n символов в строку, начинающуюся с s1, усекая или заполняя нулем строку, начинающуюся с s2, если это необходимо. Результирующая строка не может завершаться нулем, если строка, начинающаяся с s2, содержит n или более символов.

**SUBSTR**

char \*substr(pa, pb)

char \*pa, \*pb;

Находит начало первого вхождения подстроки (на которую указывает pa) в строке, на которую указывает pb. Возвращает NULL, если pa не найдено в pb. См. функции strcmp, streq и index, также описанные в этой главе, для аналогичных строковых функций.

**SWAB**

swab(s1, s2, n)

char \*s1, \*s2;

int n;

Копирует n байтов из s1 в s2, меняя местами каждую пару байтов.

**TELL**

unsigned int tell(fd)

FILE \*fd;

Возвращает байтовое смещение от начала файла, с которого начнется следующая операция ввода-вывода для этого файла. Файл определяется указателем fd на его файловый дескриптор. Если tell вызывается для файла длиной более 64 КБ, ее возвращаемое значение подлежит арифметическому переполнению. См. функции otell и rtell.

**TOLOWER**

char tolower(c)

char c;

Возвращает эквивалент c в нижнем регистре, если c является буквенным символом ASCII в верхнем регистре, в противном случае возвращает c. (K&R, стр. 145, 156.)

**TOPOFMEM**

char \*topofmem()

Возвращает CCEDATA (см. функции brk, evnbrk и sbrk).

**TOUPPER**

char toupper(c)

char c;

Возвращает эквивалент c в верхнем регистре, если c - это буквенный символ ASCII нижнего регистра, в противном случае она возвращает c (см. K&R, стр. 156).

**UBRK**

char \*ubrk(u)

unsigned u;

Возвращает указатель на область памяти размером u. Поскольку u беззнаковый, ubrk не может «вернуть» выделенную память. Он возвращает ERROR, если не может найти свободную область подходящего размера. См. функции sbrk, evnbrk, wrdbrk и brk, также описанные в этой главе.

**UGETCHAR**

ugetchar(c)

char c;

Заставляет следующий вызов getchar вернуть c. Вызов ugetchar более одного раза между последовательными вызовами getchar не повлияет на состояние стандартного ввода.

**UNGETC**

RESULT ungetc(c, fd)

char c;

STREAM fd;

Записывает символ c в самый последний прочитанный байт буфера ввода-вывода, связанный с файлом, открытым для буферизации на входе через fopen. Функция ungetc также уменьшает указатель на следующий байт, который нужно прочитать из файлового буфера ввода-вывода, чтобы он указывал на только что записанный байт.

Функция ungetc в случае успеха возвращает неопределенное значение. Вызов ungetc возвращает ERROR (-1), если она не может выполнить свою функцию: например, если указанный файл не подлежал буферизации через fopen.

Вызов ungetc для файла не имеет смысла, если для того же файла ранее не вызывались функции fgets, fscanf, getc или getw (функции ввода буферизованного файла). Только один вызов функции ungetc между вызовами функций ввода буферизованного файла для данного файла может гарантировать желаемый эффект (см. K&R, стр. 156).

**UNLINK**

RESULT unlink(fspec)

FILESPEC fspec;

Удаляет файл, указанный в fspec, из файловой системы, функция unlink возвращает SUCCESS (0), если файл был успешно удален. Функция unlink возвращает ERROR (-1) и не удаляет файл, если: (1) указанная спецификация файла неверна или (2) файл не может быть удален из-за ошибки на уровне операционной системы (см. K&R, стр. 163).

**WAIT**

RESULT wait(pid)

unsigned pid;

Блокирует выполнение процесса до завершения процесса с идентификатором процесса pid. Возвращает ERROR (-1), если такой идентификатор процесса отсутствует. Всегда немедленно возвращает значение ERROR.

**WRDBRK**

wrdbrk(u)

unsigned u;

Возвращает указатель на область памяти размера u. Поскольку u не имеет знака, wrdbrk не может "вернуть" выделенную память. Она возвращает ERROR (-1), если не может найти свободную область нужного размера. См. sbrk, evnbrk, ubrk и brk, также описанные в этой главе.

**WRITE**

int write(fd, buffer, num\_bytes)

FILE \*fd;

TYPE \* buffer;

int num\_bytes;

Возвращает число байтов, указанных в num\_bytes из зоны, на которую указывает buffer. Вывод осуществляется в файл, открытый для прямого (небуферизованного) вывода. Файл задается файловым дескриптором fd.

write возвращает фактическое количество записанных байтов. Это может быть меньше, чем num\_bytes. Если дескриптор файла недействителен или файл не может быть записан, возвращается значение ERROR (-1), указывающее на ошибку.

Каждый файловый дескриптор содержит указатель на следующую запись, доступ к которой осуществляется в операциях ввода-вывода файлов. Вызов функции write опережает этот указатель на количество записанных байтов. Последующий вызов чтения или записи начнется с новой позиции этого указателя. Вызывая seek, вы можете изменить положение этого указателя ввода-вывода файла без чтения или записи. (См. K&R, стр. 160).

**XMAIN**

xmain()

Первая функция C вызывается при запуске программы. Она устанавливает аргументы в main и выполняет перенаправление ввода-вывода, если переключатель REDIRECT установлен в CUSTOMIZ.H перед перекомпиляцией CRUNT2.C. Перенаправление ввода-вывода - это возможность перенаправлять ввод консоли, вывод консоли или аргументы команды в файлы или из файлов. Консольный ввод перенаправляется путем указания имени файла, которому предшествует '<' в командной строке. Затем консольный ввод берется из файла. Вывод консоли перенаправляется путем указания имени файла, которому предшествует '>' в командной строке. Затем консольный вывод передается в этот файл. Перенаправление аргумента команды задается путем предшествования имени файла с символом '@' в командной строке. Затем аргументы команды берутся из файла.

Аргументы командной строки передаются программе путем установки двух параметров в вызове main. Левый параметр - это количество аргументов. Правый параметр-это массив указателей на строки, по одной строке для каждого аргумента.

Если код перенаправления командной строки и ввода-вывода не требуется или требуется другое действие, можно указать программу с процедурой верхнего уровня xmain, а не main.

**XREV**

TYPE \*xrev(nargs)

TYPE nargs[];

Переделывает параметры в функциях списка. В SuperSoft C предполагается, что количество указателей находится в nargs[0], а массив параметров начинается с &nargs[1], xrev возвращает указатель на список аргументов. Обратитесь к главе 2, [*использование функций list*](#_Использование_функций_list_4), чтобы увидеть xrev в использовании.

**XRPRINTF**

xrprintf(line, args)

char \*line;

TYPE args[];

Делает всю работу для printf и др. (См. printf, описанную ранее.) Она ожидает, что массив символов (line) запишет свою выходную строку в массив аргументов (args). Первым элементом args должна быть строка format. (См. функцию printf, описанную ранее).

**XRSCANF**

xrscanf(kind, u\_kind, where, args)

int (\*kind)(), (\*u\_kind)();

TYPE args[];

Делает всю работу для scanf и др. (См. scanf, описанную ранее в этой главе). Функция xrscanf ожидает передачи двух функций. kind(where) должна возвращать байт из ввода. u\_kind(c, where) должна вернуть байт во вход. args - это строка формата (см. scanf), за которой следует массив адресов. Функцию xrscanf нельзя вызвать рекурсивно, потому что она использует глобальную статику для некоторой межфункциональной связи.

# Глава 4 Объединение исходного кода и изменение кода ассемблера

## Метод 1: Перемещающие ассемблеры, модули и библиотеки

Язык C построен так, что переменные и функции могут быть скомпилированы по отдельности в перемещаемые модули и связаны вместе для формирования рабочих программ с использованием функций, доступных в большинстве компоновщиков. (См. [*Главу 2, "Использование компилятора SuperSoft C"*.](#_Глава_2_Использование_1)) Компилятор SuperSoft C допускает класс хранения C EXTERN для внешних (глобально доступных или глобальных) переменных и функций. В частном случае функций, те, которые не были объявлены ранее, автоматически объявляются внешними. Компилятор SuperSoft C переводит класс хранилища EXTERN в соответствующую директиву ассемблера, чтобы разрешить ссылки во время компоновки.

Должно быть ровно одно определение каждой глобально доступной переменной или функции. Каждый внешний объект с таким же именем будет привязан к одному и тому же определению в программе. Различные декларации должны соответствовать определениям. Хотя. Компилятор C проверит все объявления в данном исходном файле на согласованность, компоновщик может не иметь возможности полностью проверить согласованность. Переменная может быть объявлена сколько угодно раз с явным указанием класса хранения EXTERN, но должна быть объявлена ровно один раз без класса хранения EXTERN и вне тела функции. Функция должна появляться ровно один раз с телом функции и без ключевого слова EXTERN.

Конечно, переменные должны отличаться в пределах нескольких первых символов, чтобы они казались уникальными для ассемблера или компоновщика. В случае M80 имена сокращаются до шести или семи символов в зависимости от версии. В формате CP/M-80 .REL имена ограничены семью символами. Кроме того, многие ассемблеры (включая M80) переводят символы нижнего регистра в имени в верхний регистр, эффективно уменьшая количество уникальных имен. По крайней мере, по причинам переносимости регистр не должен использоваться для различения имен переменных или функций. Например, программа не должна содержать "THelp" и "TheLP" в качестве уникальных имен переменных.

**Настоятельно рекомендуется использовать supersoft C с перемещающим ассемблером**, а не с абсолютным ассемблером, таким как ASM или ASM86. Перемещение, доступное пользователям с перемещаемым пакетом ассемблера, таким как пакет RMAC от Digital Research, на сегодняшний день является самым простым и наиболее удобным методом включения как пользовательских, так и библиотечных функций в окончательный исполняемый файл (COM, CMD или EXE).

Компоновка обычно происходит быстрее, чем компиляция. После компоновки файл можно использовать снова и снова. Компоновка также позволяет группам людей работать над программой, давая любому из этих людей возможность комбинировать и выполнять программу, но запрещая каждому доступ к источнику другого. Это последнее преимущество чрезвычайно важно в большом проекте или в небольшом проекте, завершенном с течением времени, особенно в системах, которые не имеют надлежащей защиты между пользователями.

Использование библиотек - самый простой способ объединения переменных и функций в рабочие программы. Требуется библиотекарь, а также компоновщик. В случае CP/M-80 компоновщиком может быть ELINK SuperSoft, а библиотекарем - LIB80. Вы должны скомпилировать каждый перемещаемый модуль, следовать инструкциям библиотекаря, чтобы создать библиотеку, включающую все модули, а затем указать эту библиотеку для связывания программы.

Большинство компоновщиков не делают особого различия между перемещаемым модулем и библиотекой. Любой из них может быть указан для создания исполняемого файла. Разница в основном в том, как они сочетаются. Перемещаемые модули полностью загружаются в исполняемый файл. Библиотеки состоят из нескольких перемещаемых модулей. В них ищет компоновщик, чтобы включить только те модули, которые удовлетворяют внешним ссылкам, не определенным ранее.

Компилятор SuperSoft C под CP/M-80 поставляется с библиотекой стандартных функций (LIBC.REL), созданной из файлов FORMATIO.C, STDIO.C, STDIO.H, FUNC.C, CRUNT2.C, MDEP.C, LONG.C, DOUBLE.C и ряд файлов .ASM. Параметр +SUBM компилятора C (описанный ниже) и файл SUBMIT LIBC.SUB использовался для создания этой библиотеки. Для компиляции и компоновки библиотеки использовалась версия 3.44 M80 и LIB80 от Microsoft. Эта библиотека содержит все функции, описанные в [*главе 3*](#_Глава_3_Функции_1). Вы можете добавить свои собственные функции или изменить предоставленные функции, добавив имена перемещаемых модулей в файл LIBC.SUB и скомпилировав соответствующие исходные файлы.

Если вы используете другую систему связывания, вы можете использовать или не использовать библиотеку LIBC.REL. ELINK совместим с LIBC.REL; LINK80 может и не быть. Если у вас возникли какие-либо проблемы, вам следует перестроить библиотеку, как описано ниже. К этой задаче нельзя относиться легкомысленно. Это требует значительного объема дискового пространства (в основном пространства каталогов для большого количества небольших файлов) и времени. Это может занять несколько часов.

Метод, описанный ниже, также может быть использован для создания собственных уникальных библиотек. При построении библиотеки вам, вероятно, придется сортировать модули в порядке зависимости: если модуль а ссылается на модуль в, то модуль а должен быть перед модулем в в библиотеке. Если это слишком сложно, вы можете попробовать поискать в библиотеке несколько раз, чтобы найти все ссылки. Большинство компоновщиков допускают поиск более чем одной библиотеки, поэтому добавление другой библиотеки в командную строку компоновщика является самым простым и разумным методом добавления функций в вашу библиотеку языка Си. Обратите внимание, однако, что нет необходимости компилировать исходные файлы в библиотеки. Для многих применений достаточно оставить их в качестве перемещаемых модулей. Библиотеки в основном полезны для выбора переменных или функций, а не для включения целых перемещаемых модулей в программу.

Компилятор SuperSoft C содержит метод создания библиотек, в которых каждая функция может быть отделена и загружена только в случае необходимости. Для этого укажите флаг +SUBM в командной строке как на первом (CC), так и на втором (C2 или C0D2C0D) проходах компилятора. Выходные данные второго прохода изменены: больше не выводится отдельный файл с расширением .ASM. Вместо этого это файл для каждой функции, каждый файл состоит из имени функции и расширения .MAC (в CP/M, MS-DOS и PC-DOS в этих именах удален символ '\_' и они сокращены до восьми символов. Для совместимости с собственными системами ассемблирования/компоновки). Также создается файл, который содержит директивы для формирования перемещаемых модулей (с именем .REL в CP/M-80 и .OBJ в MS-DOS). Этот файл называется $.SUB. В CP/M этот файл можно выполнить напрямую через SUBMIT, и он будет собирать соответствующие перемещаемые модули. После выполнения файла $.SUB необходимо использовать библиотекаря для включения функций в библиотеку.

Файл LIBC.SUB содержит файл SUBMIT, который будет создавать стандартную библиотеку SuperSoft C. Это не обычный файл SUBMIT, но он требует рекурсивной способности. Программа SuperSoft SH.COM (поставляемая с компилятором C для CP/M-80) допускает рекурсию и должна использоваться с LIBC.SUB.

SH похож на SUBMIT, но более мощный, чем он. SUBMIT допускает девять аргументов: от $1 до $9. SH допускает неопределенное количество аргументов: $0 содержит имя файла SUBMIT, а $1, $2, $3, ..., $9, $10, $11, ... содержат аргументы. В результате SH и SUBMIT интерпретируют $11 по-разному. SH интерпретирует его как одиннадцатый аргумент, а SUBMIT интерпретирует его как первый аргумент, за которым следует 1.

Файл SH может содержать другие строки, вызывающие SH. Каждый раз при вызове SH он складывает старый файл $$$. SUB, переименовывая его в $nn.SUB, где nn заменяется уникальным десятичным числом. Затем он помещает в конец файла SUBMIT строку вида 'SH -E $nn.SUB', которая извлекает старый файл $$$.SUB. Путем стирания $$$.SUB и переименования $nn.SUB в $$$.SUB.

Дополнительные возможности SH: (1) оператор SHIFT сдвигает список аргументов "вниз" на единицу ($1 становится $0, $2 становится $1 и т. д.). (2) SH удаляет комментарии (строки, которым предшествует ';') из файла $$$.SUB, чтобы позволить работать более крупным файлам $$$. SUB. '$\*' в файле .SUB расширяется до списка аргументов, каждый аргумент разделен пробелом, и '$:,' расширяется до списка аргументов, каждый аргумент отделен символом ','. (3) Пустые строки, неполные последние строки и большие файлы SUBMIT вызывают выполнение SUBMIT неожиданным образом; с SH такой проблемы нет. (4) SH можно использовать с любого привода. SUBMIT можно использовать только с диска A.

## Метод 2: Директива #INCLUDE

Директива препроцессора #INCLUDE может включить в вашу программу весь файл, содержащий желаемые переменные или функции. Хорошей практикой является размещение любых директив #INCLUDE для файлов функций стандартной библиотеки в преамбуле вашей программы сразу после любых определений внешних данных.

В примерах программ, поставляемых с SuperSoft C, используется этот метод построения программ из функций. Предполагается, что вы используете абсолютный ассемблер, такой как ASM CP/M-80. Использование директивы #INCLUDE является наименее распространенным знаменателем: поставляемые примеры программ будут работать либо с абсолютным ассемблером, либо с перемещаемым ассемблером. В верхней части примеров программ находятся операторы #INCLUDE для соответствующих библиотечных модулей. Например, SAMP1.C включает CRUNT2.C и FUNC.C.

Недостатком включения всех необходимых библиотечных файлов в вашу программу является то, что время компиляции вашей программы будет включать время компиляции всего включенного кода. Это также может сделать вашу программу излишне большой из-за включения в нее функций, которые никогда не вызываются. Если вы используете перемещаемый ассемблер, вы, вероятно, захотите скомпилировать примеры программ модифицированным способом. Удалите операторы #INCLUDE из примеров программ. Затем скомпилируйте, как описано в [*главе 2*](#_Глава_2_Использование_2).

В результате вы получите примерно 25 процентов от размера исполняемого файла, который вы получили бы, если бы использовали абсолютный ассемблер. Кроме того, время компиляции составляет небольшой процент от первоначального времени компиляции.

## Метод 3: Список имен файлов командной строки CC

Если вы укажете несколько имен файлов в командной строке для CC, на первом проходе компилятора, все указанные файлы будут проанализированы в указанном порядке, как если бы они были одним файлом. Вывод по-прежнему будет помещен в единственный выходной файл. Таким образом, вы можете включить в свою программу любой файл исходного кода C, просто указав его имя файла в командной строке CC. Командная строка CC, которая включает все функции стандартной библиотеки SuperSoft C в программу Y.C имеет вид:

**CC Y.C CRUNT2.C STDIO.C ALLOC.C FUNC.C FORMATIO.C**

Этот метод имеет все преимущества и недостатки предыдущего (с использованием #INCLUDE), но более гибкий. Чтобы изменить файлы, включенные в вашу программу, вместо того, чтобы изменять вашу программу, вы должны просто ввести другой набор имен файлов в командной строке CC.

## Метод 4: Предварительная компиляция и вставка в заголовочный файл

Любой из файлов библиотеки, которые находятся в исходном коде SuperSoft C, может быть скомпилирован, и полученный файл вставлен с незначительным редактированием в библиотеку времени выполнения компилятора. Это увеличивает скорость, с которой вы можете компилировать программы, включающие эти функции. Однако этот метод имеет тот же недостаток, что, возможно, чрезмерный размер программы, с двумя другими комплексными методами, описанными выше.

Конечно, вы не ограничены включением библиотек в целом. Если у вас есть представление о функциях, которые вы обычно будете использовать в своих программах, вы можете создать свою собственную версию библиотеки времени выполнения, предварительно скомпилировав и вставив эти функции в свой файл заголовка. Также не обязательно, чтобы эти функции были функциями, поставляемыми с компилятором: они могут быть созданы вами. Этот метод может сэкономить вам значительное время при компиляции ваших программ и значительное пространство для программ за счет исключения ненужных функций.

C SuperSoft позволяет пользователю добавлять код в файл заголовка генератора кода, файл, который автоматически помещается в начало программ C. (Заголовок предоставляет необходимые машинно-зависимые функции.) Любую функцию, уже находящуюся в исходном коде ассемблера и включенную в файл заголовка, не нужно перекомпилировать и повторно оптимизировать каждый раз при компиляции остальной части программы. Это особенно полезно при использовании абсолютного ассемблера, такого как ASM CP/M-80 или ASM86 CP/M-86.

Чтобы использовать этот метод, пользователь компилирует необходимые процедуры, создавая файл на языке ассемблера. Этот исходный файл ассемблера затем добавляется в файл заголовка.

Ниже приведено подробное, шаг за шагом, описание того, как файл подпрограмм CRUNT2.C может быть перемещен в подпрограммы времени выполнения под CP/M-80, таким образом устраняя необходимость использования директивы #INCLUDE "CRUNT2.C" (или ссылки на объектный файл CRUNT2) .

1. Мы готовы скомпилировать CRUNT2.C. Выполните следующие команды:

**A>CC CRUNT2.C**

**...**

**A>C2 CRUNT2.COD +ASM +ZA -RH -RT**

Опция '+ZA' указывает генератору кода, чтобы все метки начинались с буквы 'A'. Это позволяет избежать любого конфликта с кодом, создаваемым обычными компиляциями, которые начинают каждую метку с буквы 'С'. Параметры -RT и -RH предотвращают включение файлов заголовка и трейлера.

1. Затем переименуйте файл в CRUNT2.LIB в рамках подготовки к слиянию с файлом C2.RH:

**REN CRUNT2.LIB=CRUNT2.ASM**

1. Перед объединением файла CRUN72.LIB необходимо выполнить небольшую правку. Введите следующее:

**A>ED CRUNT2.LIB**

**\*4a**

**\*3T**

тогда вы увидите

**;C Optimizer V1.2**

**ORG 256**

**;C Compiler V1.2**

Когда вы перечисляете верхнюю часть файла на своем экране, ваш файл должен быть таким же (за исключением номеров версий), как и указанный здесь. Теперь вы должны удалить следующую строку:

**ORG 256**

1. После удаления этой строки выйдите из редактирования, нажав 'E' (таким образом, сохраните свои изменения).

## Метод 5: Вырезать и вставить

Этот метод прост по идее, но трудоемок в исполнении. Идея состоит в том, чтобы создать из файлов функций стандартной библиотеки файл или файлы, содержащие только те данные и определения функций, которые требуются вашей программе (вырезать), а затем объединить файлы, созданные таким образом, в вашу программу перед ее компиляцией (вставка).

Преимущество этого подхода заключается в том, что исходный и объектный код вашей программы будут настолько малы, насколько позволяет реализация вашей программы. Конечно, вы должны быть осторожны, чтобы все данные и определения функций, необходимые для выполнения вашей программы, присутствовали и не были повреждены. Ясно, что это может занять много времени, и в процессе могут закрасться некоторые неприятные ошибки.

Однако в некоторых приложениях может быть важно, чтобы ваши программы максимально эффективно использовали доступную память. Размер одних программ может приближаться к объему памяти вашей системы или превышать его, в то время как другие могут потребовать соответственно большие объемы хранения данных во время выполнения.

Вы также можете неоднократно использовать определенное подмножество стандартных библиотечных функций в сочетании с некоторыми собственными функциями. В этой ситуации вы можете использовать этот метод в качестве первого шага к созданию собственного настраиваемого набора функций стандартной библиотеки SuperSoft C.

Вам не следует пытаться использовать этот метод, если вы не знакомы с C и вашей собственной операционной системой. Далее следует лишь очень общий план шагов, необходимых для метода вырезания и вставки.

1. Сделайте копию на диске каждого файла, содержащего определения данных или функций, которые вы хотите включить в свою программу.
2. Используя ваш редактор, удалите из каждого файла те данные и определения функций, которые вы не хотите включать в свою программу. Будьте осторожны, чтобы не удалить определения тех функций, которые вызываются функциями, которые вызывает ваша программа, или любые определения данных, требуемые любой функцией, которую вы собираетесь включить.
3. Если теперь у вас есть несколько файлов, содержащих интересующие вас определения, вы можете оставить их как отдельные файлы или объединить их по своему усмотрению.
4. Вы снова столкнетесь с выбором того, как встроить созданные вами файлы в свою программу. Вы можете использовать любой из описанных ранее методов для включения файлов функций стандартной библиотеки SuperSoft C. Теперь ваши программы будут содержать только те данные и определения функций, которые им необходимы. Другой вариант - использовать ваш редактор для вставки созданных вами определений в свою программу в соответствующие места.
5. Теперь вы должны отредактировать файл C2.RH. Ваш экран должен показать:

**A>ED C2.RH**

**\*#A**

**\*F\*\* встаьте любого пользователя ^Z0lt**

**; \*\*\*\*\* вставьте сюда любой код пользователя \*\*\*\*\***

**\*rcrunt2.lib**

**\*e**

Это заставит редактор читать файл CRUNT2.LIB.

После выхода из редактора с помощью команды "E" все функции, содержащиеся в CRUNT2.C, будут перемещены в системный исполняемый файл C2.RH. Это означает, что вам больше не нужно будет использовать директиву #INCLUDE "CRUNT2.C". Однако, поскольку вы всегда будете загружать эти функции, даже если они не нужны, может потребоваться другой C2.RH для разных программ.

## Как переориентировать код, сгенерированный компилятором

Перемещающий ассемблер освобождает программистов от заботы о фактических местах, в которых их программы выполняют или извлекают данные. Он делает это, генерируя объектный код, в котором адреса памяти отображаются как смещения из некоторой относительной точки. Эти относительные точки могут быть либо внешне определенными ссылками, либо относительными исходными точками программы. Внешняя ссылка использует символическую метку, ссылающуюся на местоположение в другом модуле. Например, внешняя ссылка в модуле A - это способ ссылаться на местоположение в модуле B, не зная абсолютного адреса этого местоположения. Относительное начало программы - это начало конкретного модуля, относительно которого вычисляются все его внутренние адреса. Относительные адреса фиксируются, когда программа связана.

Ассемблеры будут поддерживать некоторые способы указания абсолютного (или относительного) происхождения вашего кода. В случае абсолютного ассемблера CP/M, ASM, инструкция ORG выполняет эту функцию. Аргумент оператора ORG используется ASM в качестве источника для кода, которому он предшествует - до следующего оператора ORG или конца файла. Команда ORG также может использоваться для установления различных относительных и абсолютных источников для каждого сегмента программы или области данных. Эта возможность особенно важна, когда вы хотите загрузить свою программу в ПЗУ, а данные - в ОЗУ. Если программа, которую вы хотите загрузить в ПЗУ, требует области данных для записи, в вашем файле должны появиться по крайней мере два оператора ORG: один в начале вашей программы, указывающий ее абсолютное происхождение в ПЗУ, а другой - в начале области записываемых данных, определяющий ее абсолютное происхождение в ОЗУ.

Окончательный вывод ассемблерного кода, сгенерированный этим компилятором, содержит два оператора ORG, которые появляются (в начале программы и в областях данных соответственно) следующим образом:

ORG \* ; Программа переориентирована здесь

...

ORG \* ; Данные переориентированы здесь

Аргумент '\*' обозначает текущее местоположение, но оператор ORG не активируется, пока вы не вставите значения для этих аргументов, соответствующие вашей ситуации.

# Приложение A Различия между SuperSoft C и стандартным C

Компания SuperSoft стремится реализовать полный язык C. Все покупатели SuperSoft C получат уведомление по мере появления обновлений. Стандартные функции языка C еще не реализованы: объявления TYPEDEF; объявление и использование битовых полей; инициализации. Статические декларации распознаются; однако они не имеют начального значения. В противном случае EXTERN STATIC работают должным образом, а локальные STATIC не сохраняют свои значения при вызовах функций.

Могут быть определены переменные LONG, FLOAT, DOUBLE и LONG FLOAT. Эти типы данных могут использоваться в любом объявлении, кроме как формальный аргумент функции. Это включает объявление STRUCT, UNION и массивов. Оператор адреса - это единственный доступный оператор для этих типов данных. Обратите внимание, что есть библиотечные функции (DOUBLE.C и LONG.C), которые работают с этими типами данных.

Компилятор SuperSoft C ожидает передачи параметров в функции в стек в порядке справа налево, за которым следует адрес возврата (предположительно, по команде вызова машины). В языке, как определено в K & R, допускается порядок справа налево или слева направо, однако, ряд компиляторов передают аргументы в порядке, обратном SuperSoft C. В частности, исходная реализация DEC PDP-11 перемещает слева направо.

Все формальные аргументы функции должны быть объявлены внутри этой функции. А именно:

func (aa, bb, cc)

int bb;

int aa, cc;

{

... принято, но

}

func (aa, bb, cc)

{

... выдаст ошибку.

Генератор кода еще не присоединяет никаких уникальных префиксов или суффиксов к именам переменных. Таким образом, глобальные идентификаторы в исходном коде C могут конфликтовать с именами и ключевыми словами ассемблера. Поэтому вы должны избегать использования в исходном коде языка Си любых зарезервированных символов или ключевых слов вашего ассемблера. Этот недостаток будет исправлен в будущем выпуске.

Параметризованные директивы препроцессора #DEFINES и #IF (с выражением) пока не поддерживаются. В отличие от стандартного C, строки сканируются только один раз для макроподстановок #DEFINE. Это означает, что во время предварительной обработки невозможно войти в неопределенный цикл. Однако это также означает, что порядок #DEFINE имеет значение. например, в

#define x y

#define y z

x будет заменен на y, а y будет заменен на z. Однако в

#define y z

#define x y

и y, и x будут заменены на z.

В SuperSoft C не должно быть промежуточных символов новой строки между меткой и связанным с ней символом ':'. Объявления массивов могут содержать только ограниченные выражения. В частности, нельзя использовать SIZEOF и выражения в скобках. Из-за способа, которым SuperSoft C анализирует локальные объявления, переменная, объявленная как REGISTER, не будет на самом деле ссылаться на регистр, если это не первое объявление REGISTER, и оно объявлено само по себе. Таким образом, следующее будет объявить x как истинную регистровую переменную, но y будет объявлен как обычная локальная:

funct()

{

register char x;

register char y;

Следующее не приведет к успешному объявлению какой-либо истинной регистровой переменной:

funct()

{

register char x, y;

Функции dup и fork, доступные в UNIX, не реализованы, в основном из-за неудобств одно процессных операционных систем. Ссылка работает иначе, чем ссылка в UNIX. Функции fopen и fdopen имеют дополнительный параметр по сравнению с реализацией UNIX, что позволяет более четко использовать буферизованный ввод-вывод. Это не создает проблем, если дополнительный параметр используется в большинстве реализаций UNIX C, поскольку последний, дополнительный параметр игнорируется.

Версия 1.2 SuperSoft C имеет более полную совместимость с UNIX, чем предыдущая версия (1.1). Больше нет ограничений на размер записи для файлового ввода-вывода. Это повлияло на read() и write(). Также был добавлен параметр режима в creat() и параметр arg0, добавленный в execl(), чтобы привести SuperSoft C в соответствие с UNIX. Этот выпуск C также содержит гораздо более полный набор директив препроцессора.

# Приложение B Доступные в настоящее время компьютеры и ОС Конфигурации компилятора SuperSoft C

Текущие операционные системы: CP/M-80, MP/M-80, CP/M-86, Concurrent CP/M-86, CP/M +, CP/M-80 3.0, MP/M-86 (и совместимые); PC-DOS, MS-DOS (и совместимый); UNIX, XENIX (и совместимые); и Central Data ZMOS.

Текущие хост-процессоры или целевые процессоры: Intel 8080, Intel 8085, Intel 8086, Intel 8088, Intel 186, Intel 286, Zilog Z80, Zilog Z8001, Zilog Z8002, Zilog Z8003 и Zilog Z8004.

SuperSoft предоставляет компилятор для любой допустимой комбинации хоста, целевой системы и операционной системы. Тем, кто заинтересован в таких системах, следует связаться с SuperSoft для получения информации о наличии.

# Приложение C Некоторые общие проблемы и решения

|  |  |  |
| --- | --- | --- |
| 1. | Проблема: | Во время ассемблирования или компоновки функция не определена |
|  | Решение: | Не забудьте включить файлы, содержащие необходимые функции. См. [*Главу 4*](#_Глава_4_Объединение_2). |
| 2. | Проблема: | Во время ассемблирования или компоновки некоторые переменные не определены. |
|  | Решение: | Не забудьте объявить все переменные. |
| 3. | Проблема: | Во время ассемблирования или компоновки функция или переменная помечены как ошибки повторяющихся определений. |
|  | Решение: | Вы случайно определили функцию или переменную более одного раза? Вы использовали имя, которое использует служба поддержки C? Сколько значащих букв (ограничение на усечение) использует ваш ассемблер? |
| 4. | Проблема: | Во время ассемблирования отображается "P" или ошибка фазы. |
|  | Решение: | Переменная или функция были определены более одного раза. |
| 5 | Проблема: | Во время сборки символ обозначается как public и external одновременно. |
|  | Решение: | Эта переменная или функция написаны с ошибками, и ассемблер не улавливает их, потому что разница - это изменение верхнего и нижнего регистра или разница, превышающая предел усечения ассемблера. Например, TPuts и Tputs могут быть одинаковыми для ассемблера, очень длинное имя и очень длинное имя также могут быть одинаковыми. |
| 6. | Проблема: | Во время компоновки не обнаруживается символ, который должен быть в библиотеке. |
|  | Решение: | Просканируйте библиотеку дважды во время компоновки. Если символ будет найден, вам необходимо изменить порядок библиотеки. Между тем, двойного сканирования должно хватить. Если символ все еще не найден, значит, символа нет в библиотеке. |
| 7. | Проблема: | alloc0 не определен во время компоновки |
|  | Решение: | Вы вызываете функции без их "совпадающего" предшественника. Например, вы вызываете fclose без fopen или free без malloc. |
| 8. | Проблема: | Выполнение программы на C не может быть прекращено с Ctl-C под CP/M. |
|  | Решение: | Это связано с тем, что CP/M обычно не управляется прерываниями и не может реагировать на символы с клавиатуры в произвольных обстоятельствах. Однако во время вывода на консоль будет работать нажатие Ctl -S, а затем Ctl -C. |
| 9. | Проблема: | Компоновщик умирает, останавливает вашу систему или сообщает, что не хватает места. |
|  | Решение: | Найдите дополнительную память или смените компоновщик. В рамках CP/M-80 мы предлагаем дисковый компоновщик SuperSoft ELINK, который не должен исчерпывать пространство. |

# Приложение D Местонахождение поставляемых функций и внешних переменных

**ALLOC.C**

|  |  |  |  |
| --- | --- | --- | --- |
| alloc | free | malloc | calloc |
| realloc | isheap |  |  |

**C2.RH, C2.RT, C2I86.RH, C2I86.RT, MDEP.C, C2I86.RTB, C2I86.RTM, C2PRE.ASM, C2POST.ASM, C2.RTM**

|  |  |  |  |
| --- | --- | --- | --- |
| bdos | bios | brk | ccall |
| ccalla | exit | inp | inp16 |
| longjmp | outp | outp16 | reset |
| setexit | setjmp | streq | errno |
| comlen | comline | ccexit |  |

**CRUNT2.C**

|  |  |  |  |
| --- | --- | --- | --- |
| envbrk | getchar | getc | isalpha |
| isdigit | islower | isupper | iswhite |
| movmem | putchar | putc | sbrk |
| setmem | strlen | toupper | ubrk |
| assert | ugetchar | wrdbrk | xmain |

**FUNC.C**

|  |  |  |  |
| --- | --- | --- | --- |
| abs | absval | atoi | perror |
| getval | index | initb | initw |
| isalnum | isascii | iscntrl | isnumeric |
| isprint | ispunct | isspace | kbhit |
| min | max | pause | peek |
| poke | putdec | qsort | rand |
| rindex | sleep | srand | strcat |
| strcmp | strcpy | strncat | strncmp |
| strncpy | substr | tolower | topofmem |

**STDIO.C**

|  |  |  |  |
| --- | --- | --- | --- |
| close | cpmver | creat | exec |
| execl | fabort | fclose | fflush |
| fgets | fopen | fputs | fread |
| fwrite | getc | getw | access |
| link | open | otell | pgetc |
| pputc | putc | putw | read |
| rename | rtell | seek | tell |
| ungetc | unlink | write | wait |
| look | nice | swab | isatty |
| mktemp | clearerr | freopen | fdopen |
| isfd | ferror | fileno | fgetc |
| fputc | get2b | put2b | chmod |

**FORMATIO.C**

|  |  |  |  |
| --- | --- | --- | --- |
| fprintf | fscanf | printf | scanf |
| sprintf | sscanf | xrprintf | xrscanf |
| xrev |  |  |  |

**DOUBLE.C**

|  |  |  |  |
| --- | --- | --- | --- |
| badd | bsub | bdiv | bmul |
| bmodulo | bxtofy | s2bcd | bcd2s |
| bneg | buneg | babs | buabs |
| bint | bentier | blog | bsqr |
| bexp | bfac | bsin | bcos |
| btan | barctan | bsign | blt |
| bgt | beq | bne | ble |
| bge | btest | bseterrflg | bmant |
| bexpo | bmov | btound | int2bcd |
| bcd2int | bcheby |  |  |

**LONG.C**

|  |  |  |  |
| --- | --- | --- | --- |
| cc1prt | cc1neg | cc1ink | cc1dec |
| cclcom | cc2mov | cc2add | cc2com |
| cc2or | cc2xor | cc2cmp | cc2mul |
| cc2imul | cc2iadd | cc2and | cc2sub |
| cc2or | cc2neg | cc3add | cc3and |
| cc3or | cc3xor | cc3sub | cc3div |

# Приложение E Использование оверлеев в CP/M-80

ELINK от SuperSoft (который доступен за дополнительную плату для компилятора C) позволяет создавать наложенные программы из существующих перемещаемых модулей C и ассемблера. ELINK - это дисковый автономный редактор и загрузчик многосегментных (оверлейных) связей. ELINK создает исполняемые сегменты, которые работают под CP/M-80 и могут обрабатывать перемещаемые файлы, созданные различными ассемблерами, включая M80 (Microsoft) и RMAC (Digital Research). Перемещаемые файлы можно собирать в библиотеки и искать. Большие программы (до 64 КБ) могут быть связаны даже в системах CP/M с памятью менее 64 КБ. Даже более крупные программы могут быть разделены на сегменты и наложены друг на друга. Наложенные программы устраняют многие ограничения, связанные с хранением в ваших программах, и делают это без цепочки или изменения кода.

Создание наложений - довольно автоматический процесс. Исходный код C не нужно изменять, чтобы учесть различия между вызовом функции в том же наложении и вызовом функции в другом наложении. Структура оверлея определяется исключительно командами ELINK, которые разделяют модули на оверлеи и корневой исполняемый файл .COM. Структура наложения может выглядеть так (следуя примеру в документации ELINK):

-------------

level 0 (root) | cmain.rel |

-------------

/ \

/ \

/ \

------------ ------------

level 1 | cov1.rel | | cov2.rel |

------------ ------------

/ \

/ \

/ \

------------- -------------

level 2 | cov21.rel | | cov22.rel |

------------- -------------

Оверлейная структура - это древовидная структура, в которой важны уровни. Каждый оверлей (представленный рамкой на приведенной выше диаграмме) может содержать любое количество локальных функций, которые будут доступны ему и его дочерним элементам. Каждая функция в оверлее может вызывать любое количество функций из других оверлеев. У вас может быть до 85 наложений с глубиной до 255 уровней.

Наложения имеют дополнительное преимущество, требуя дисциплины при структурировании модульности программы. Древовидная структура, налагаемая наложениями, к счастью, идеально подходит для поддержки ряда типов систем, таких как программы, управляемые с помощью меню. Оверлеи могут точно отражать структуру программы.

Помещая библиотеки C в корень, можно добиться значительной экономии, поскольку они расположены только в одном месте и не должны загружаться с каждым оверлеем.

ELINK не проверяет путь вызова и предполагает, что родитель в дереве наложения находится вместе с дочерним. Это просто означает, что каждая структура наложения может вызывать только подпрограммы на всем пути к корню дерева. Вы должны избегать того, чтобы функция выполняла автоматический вызов загрузки функции в другом наложении на том же уровне структуры. Это приведет к тому, что возвращаемая функция станет нерезидентной, поскольку вызываемая функция перекрывает вызывающую.

ELINK содержит модифицированные C2.RH и C2.RT, которые используют различные функции ELINK и необходимы для наложения. Изменения очень просты и касаются только расположения верхней части пользовательской программы. (ALLOC и некоторые другие функции не будут работать должным образом без этих измененных файлов.)

Чтобы создать и выполнить наложенную программу, выполните следующие действия.

1. Скомпилируйте все файлы .C с помощью SuperSoft C и соберите с помощью M80. (RMAC также может работать, но у нас нет подтверждения этого.)
2. Со всеми файлами ".REL" на текущем диске введите:

ELINK so(cex);

Это заставит ELINK читать CEX.LNK для команд, которые создадут структуру наложения, определенную выше. (Дополнительную информацию см. В документации ELINK).

CMAIN.COM и CMAIN.OVL созданы. Файл .COM должен содержать корневую и С библиотеки вместе с пакетом времени выполнения ELINK. Источник этой среды выполнения включен в ELINK на случай, если вы хотите настроить свою среду. CMAIN.OVL содержит весь код оверлея в абсолютной форме вместе с каталогом оверлеев в файле. CP/M 2.2 требуется для наложений отдельные файлы и точки входа АВТОМАТИЧЕСКОЙ загрузки. Прямое наложение файлов поддерживается в CP/M 1.4. См. Руководство ELINK для получения более подробной информации.

Просто набрав CMAIN, программа запустится. Оверлеи будут вызываться по мере необходимости.

# Приложение F Сообщения об ошибках

**"A formal arg is not declared" (CC)**

Объявление функции, предшествующее этой ошибке, имеет формальный аргумент без соответствующего объявления типа.

Например:

**func(a, b)**

**char \*a;**

**{**

**}**

будет генерировать эту ошибку.

**"Already defined" (CC)**

Это функция, формальный аргумент, локальный (автоматический) или структурный элемент, который был определен ранее.

**"Array or pointer being lengthened" (CC)**

Это объявление увеличивает заявленный размер массива, на который ссылаются. В случае указателя это означает, что объект, на который он указывает, будет казаться больше, чем раньше, в результате чего любой указатель будет умножаться или делиться на новый размер. Это не всегда является ошибкой, так как обычно объявляются внешние объекты произвольного размера:

**extern char arr[];**

а затем в другом месте объявите определенный размер:

**char arr [99];**

**"Bad break" (CC)**

break был обнаружен вне всех циклов (for, while, do или switch).

**"Bad register op" (C2 или COD2COD)**

Ошибка U-кода. Регистр используется в коде операции, который не имеет режима регистра.

**"Bad register struct or union" (CC)**

Регистровая переменная не может быть структурой или объединением, но ее размер в настоящее время должен быть не больше int.

**"Bad register type" (CC)**

Слишком сложное объявление применяется к регистру. Например:

register int arr[99];

невозможно.

**"Bad usage" (CC)**

Такое использование переменной несовместимо с ее объявлением.

**"Can't open include file" (CC)**

Была обнаружена директива препроцессора #include, но указанный файл не существует.

**"Can't subscript" (CC)**

Попытка присвоить индекс любому выражению (включая переменную), которое не является массивом или указателем.

**"Can't take address" (CC)**

Попытка указать адрес выражения

**"Can't write to output file" (CC, C2 или COD2COD)**

Попытка записи в выходной файл, но произошла ошибка ввода-вывода. Обычно указывает, что файловая система вывода заполнена.

См. параметр -ofile для решения проблемы.

**"Cast" (CC)**

Попытка использовать приведения. Может не сработать.

**"Continue without matching loop" (CC)**

Обнаружено продолжение за пределами окружающего оператора for, while или do.

**"Declaration mismatch" (CC)**

Повторное объявление переменной отличается от предыдущего объявления. Это сообщение не появится, если переменная повторно объявлена идентично предыдущим объявлениям.

**"Divide by 0" (C2 или COD2COD)**

Во время оптимизации (сворачивание констант) обнаружено деление на ноль. Ваш алгоритм не должен требовать деления на ноль.

**"Don't add pointers" (CC)**

Попытка добавить указатель (или массив) к указателю (или массиву). Это неправильное использование арифметики указателей, которая предполагает, что одна сторона двоичного оператора будет чистым значением.

**"Don't negate pointers" (CC)**

Обнаружено подвыражение вида int-указатель. Возможно, вы имели в виду указатель - int.

**"Duplicate default" (CC)**

В контексте текущего switch появилось несколько меток по умолчанию. Каждый вложенный переключатель может иметь одно значение по умолчанию.

**"Expecting formal arg" (CC)**

Эта ошибка появляется, когда внутри объявления списка аргументов функции ожидается идентификатор. То есть был замечен знак '(' или ','.

**"Expecting function body" (CC)**

Были объявлены все формальные аргументы для этого объявления функции. На этом этапе ожидается появление первого символа '(' в теле функции.

**"Expecting ',' or ')'" (CC)**

Внутри списка формальных аргументов функции ни ',' ни ')' не было найдено, когда ожидалось.

**" Expecting ',' or ';'" (CC)**

За объявлением типа формального аргумента не следовало ',' или ';'. Иногда указывает на путаницу со стороны компилятора в отношении ваших намерений в объявлениях. Похоже, что синтаксис C оставляет мало возможностей для исправления ошибок в объявлениях.

**" Expecting ')' or ','** **in function call" (CC)**

Относится к плохо сформированному вызову функции.

**"Expecting type declaration" (CC)**

Ожидается объявление элемента структуры. Это должно начинаться с объявления базового типа, такого как int или char.

**"File close error" (CC, C2 или COD2COD)**

Ошибка вывода произошла во время записи нескольких последних байтов в выходной файл или во время фактического закрытия. Обычно указывает на то, что в выходной файловой системе недостаточно места.

См. Параметр -ofile для решения проблемы.

**"gen err t1" (C2I86)**

Обнаружен неверный U-код. Либо C2I86 считал неверный U-код, либо возникла проблема с внутренней памятью C2I86.

**"gen err t2" (C2I86)**

Обнаружен неверный U-код. Либо C2I86 считал неверный U-код, либо возникла проблема с внутренней памятью C2I86.

**"gen err t3" (C2I86)**

Обнаружен неверный U-код. Либо C2I86 считал неверный U-код, либо возникла проблема с внутренней памятью C2I86.

**"gen err t6" (C2I86)**

Обнаружен неверный U-код. Либо C2I86 считал неверный U-код, либо возникла проблема с внутренней памятью C2I86.

**"gen err t7" (C2I86)**

Обнаружен неверный U-код. Либо C2I86 считал неверный U-код, либо возникла проблема с внутренней памятью C2I86.

**"Internal error: close on bad fd" (CC)**

Закрытие включаемого файла, которое должно было пройти, не удалось. Это никогда не должно происходить и обычно означает, что стек компилятора был перезаписан. Добавьте больше памяти в вашу систему или уменьшите количество объявлений в этой компиляции (например, разделив программу на несколько файлов, каждый из которых компилируется отдельно). Также см. параметр -bufsiz для уменьшения размеров буфера ввода-вывода компилятора.

**"Internal error: extra free" (CC)**

Выражение оставило подвешенное подвыражение. Не должно случиться. Возможно означает, что куча компилятора была перезаписана. Добавьте больше памяти в вашу систему или уменьшите количество объявлений в этой компиляции (например, разделив программу на несколько файлов, каждый из которых компилируется отдельно). Также см. параметр -bufsiz для уменьшения размеров буфера ввода-вывода компилятора.

**"Internal error: missing free" (CC)**

Выражение содержало ложное подвыражение. Не должно случиться. Возможно означает, что куча компилятора была перезаписана. Добавьте больше памяти в вашу систему или уменьшите количество объявлений в этой компиляции (например, разделив программу на несколько файлов, каждый из которых компилируется отдельно). Также см. параметр -bufsiz для уменьшения размеров буфера ввода-вывода компилятора.

**"Internal error: optdel" (C2 или COD2COD)**

Была сделана попытка удалить U-код из недопустимого места во внутренней памяти оптимизатора. Такого никогда не должно быть. Может означать, что произошло переполнение памяти. Уменьшите размер самой большой функции или используйте параметр -o. Также см. параметр -bufsiz для уменьшения размеров буфера ввода-вывода компилятора.

**"Internal warning: basic block" (C2 или COD2COD)**

Игнорируйте это предупреждение:. указывает на незначительную несогласованность в базовой обработке ошибок оптимизатора.

**"inval pseudo op type" (C2I86)**

Обнаружен неверный U-код. Либо C2I86 считал недопустимый U-код, либо есть проблема с внутренней памятью C2I86.

**"Line too long" (CC)**

Исходная строка слишком длинная или строка становится слишком длинной после предварительной обработки. То, что "видит" компилятор, печатается с опцией +1.

**"Missing bracketing symbol" (CC)**

Символ скобок, обычно ']', ')' или '}', отсутствует. Точный символ печатается в выходном файле.

**"Missing '}'" (CC)**

Был замечен конец файла до того, как был замечен последний '}'. Это коварная ошибка, так как она может включать в себя дополнительный '{' или отсутствующий '}', отличные от последнего '}'. Проверьте правильность закрытия всех предыдущих составных операторов.

**"Missing ':'" (CC)**

отсутствует ':' при попытке разобрать троичный оператор if, '?'

**"Missing label" (CC)**

В заявлении goto отсутствовала метка.

**"Missing quote or apostrophe" (CC)**

Читается строка, у которой нет завершающей кавычки или читается символьный литерал, у которого нет завершающего апострофа.

**"Missing ';'" (CC)**

В конце оператора или объявления должна стоять точка с запятой.

**"Missing while" (CC)**

Отсутствует while в конце оператора do.

**"Must be a constant" (CC)**

Ожидается константа в объявлении массива. В настоящее время sizeof не допускается в объявлениях массивов. Выражения в скобках в настоящее время не допускаются. Переменные никогда не будут разрешены.

**"Must be lvalue" (CC)**

На этом этапе ожидается допустимая левая часть. Левая часть должна иметь адрес и иметь возможность хранить переменное значение. Обратите внимание, что имя массива не может быть присвоено, и, следовательно, это не лучшая левая часть.

**"Nonsensical pointer operation" (CC)**

Операция с указателем, которую вы выполняете, обычно не считается допустимой или разумной, но она все равно будет выполнена.

**"No symbol table room" (CC)**

Куча компилятора заполнена или была перезаписана. Добавьте больше памяти в вашу систему или уменьшите количество объявлений в этой компиляции (например, разделив программу на несколько файлов, каждый из которых компилируется отдельно). Также см. Параметр -bufsiz для уменьшения размеров буфера ввода-вывода компилятора.

**"Not a label" (CC)**

Попытка выполнить переход к чему-то, что не является меткой

**"Not an array or pointer" (CC)**

Попытка присвоить индекс переменной, не являющейся массивом или указателем.

**"Not a pointer or array" (CC)**

Попытка использовать оператор косвенного обращения '\*' к чему-либо, кроме указателя или массива.

**"Not a pointer to a function" (CC)**

Выполняется попытка вызова функции, но вызываемая функция ранее была объявлена не как функция. Например, следующее сообщение вызовет это сообщение:

**int fn;**

**fn();**

**"Not a struct or union element" (CC)**

Попытка использовать '.' или оператор '->', но правая часть операции не является элементом структуры или объединения.

**"Not declared" (CC)**

Попытка использовать необъявленную переменную. Это сообщение об ошибке появится только при первом таком появлении переменной.

**"Opcode error" (C2 или COD2COD)**

Либо неверный U-код был введен в C2 (или COD2COD), либо оптимизатор сгенерировал неверный код операции: скорее всего, внутренняя ошибка.

**"Open failure" (CC, C2I86, COD2COD или C2)**

Не удается открыть выходной файл. Такая ошибка может быть вызвана, если файловая система не существует, если она неправильно настроена (т.е. Нет диска в приводе), если на диске нет места или неправильно сформировано имя файла.

**"Operation on incompatible pointers" (CC)**

В арифметической операции используются два указателя, но они указывают на разные объекты.

**"Optimizer table overflow" (C2 или COD2COD)**

Была сделана попытка добавить U-код во внутреннюю память оптимизатора, но места не было. Это означает, что оптимизатору не хватает памяти. Уменьшите размер самой большой функции или используйте параметр -o. Также см. параметр -bufsiz для уменьшения размеров буфера ввода-вывода компилятора.

**"Optout empty" (C2 или COD2COD)**

Попытка выдать U-код из внутренней памяти оптимизатора, но ничего не найдено. Этого никогда не должно происходить, и даже если это произойдет, это не должно повлиять на правильность сгенерированного кода.

**"Out of heap" (CC)**

Обычно это означает, что куче компилятора не хватает места или она была перезаписана. Добавьте больше памяти в вашу систему или уменьшите количество объявлений в этой компиляции (например, разделив программу на несколько файлов, каждый из которых компилируется отдельно). Также см. параметр -bufsiz для уменьшения размеров буфера ввода-вывода компилятора.

**"psu err t4" (C2I86)**

Обнаружен неверный U-код. Либо C2I86 считала неверный U-код, либо возникла проблема с внутренней памятью C2I86.

**"psu err t5" (C2I86)**

Обнаружен неверный U-код. Либо C2I86 считала неверный U-код, либо возникла проблема с внутренней памятью C2I86.

**"Redeclaration of a label" (CC)**

Метка была объявлена более одного раза внутри одной функции. Метки объявляются просто следующим за ними знаком ':'. Вам, вероятно, в любом случае не следует использовать метки.

**"Redeclaration of struct type" (CC)**

Тип структуры был объявлен более одного раза:

struct x {

struct x {

int i;

char j;

}

}

Выше x был повторно объявлен.

**"Redefined" (CC)**

Идентификатор #defined был повторно объявлен. Это сообщение появится только в том случае, если новое объявление - это строка, отличная от старой.

**"Registers have no address" (CC)**

Была сделана попытка взять адрес (оператор '&') регистровой переменной. Это сообщение об ошибке появится тогда и только тогда, когда регистровая переменная является истинной регистровой переменной.

**"shll error" (C2 или COD2COD)**

Обнаружен неверный U-код. Либо C2I86 считал неверный U-код, либо возникла проблема с внутренней памятью C2I86.

**"String size exceeded" (CC)**

Строка больше, чем может обработать компилятор. Строка может быть незавершенной. В противном случае попробуйте использовать strcat во время выполнения.

**"Too many '&'s" (CC)**

Попытайтесь взять адрес, используя оператор '&', адреса.

**"Undefined struct" (CC)**

За ключевым словом struct следует идентификатор, который никогда не был определен.

**"Unrecognized '#'" (CC)**

За # следует директива, которая не распознается. Допустимые следующие директивы: if, ifdef, ifndef, else, endif, include и define.

**"Variable or constant expected" (CC)**

На этом этапе программы ожидалась переменная или константа. Скорее всего, было использовано плохо сформированное выражение.

# Приложение G Функции С плавающей запятой двойной точности

Функции DOUBLE.C можно разделить на четыре группы:

**Группа 1** содержит три арифметических функции операндов: сложение, вычитание, умножение, деление и модуль (остаток). Эти функции возвращают вычисленное значение в своем первом аргументе. Возвращаемое значение - это адрес этого первого аргумента.

Функции **группы 2** по большей части представляют собой функции с двумя аргументами, которые возвращают вычисленное значение во втором аргументе. Эти команды преобразовываются в двойную строку, строку в двойную, целочисленное усечение, синус, косинус, тангенс, квадратный корень, возведение в степень, натуральный логарифм, отрицание, абсолютное значение и факториал. Радианы используются для функций, требующих или возвращающих измерение угла.

**Группа 3** содержит, по большей части, две функции условного операнда и функцию проверки. Эти функции возвращают TRUE (1), если данное условие выполняется между аргументами, в противном случае - FALSE (0). Команды группы 3 равны, не равны, больше, больше или равны, меньше и меньше или равны. Также доступна тестовая функция, которая возвращает POSITIVE (1), ZERO (0), или NEGATIVE (-1) или в зависимости от аргумента. Эти функции не устанавливают Berrflag.

**Группа 4** содержит разные функции.

**Группа 1**

**double \***

**Badd(dest, arg1, arg2) double \*dest, \*arg1, \*arg2;**

Сложение трех операндов:

\*dest = \*arg1 + \*arg2;

**double \***

**Bsub (dest, arg1, arg2) double \*dest, \*arg1, \*arg2;**

Вычитание трех операндов:

\*dest = \*arg1 + \*arg2;

**double \***

**Bdiv(dest, arg1, arg2) double \*dest, \*arg1, #arg2;**

Деление трех операндов:

\*dest = \*arg1 / \*arg2;

**double \***

**Bmul (dest, arg1, arg2) double \*dest, \*arg1, \*arg2;**

Умножение трех операндов:

\*dest = \*arg1 \* \*arg2;

**double \***

**Bmodulo(dest, arg1, arg2) double \*dest, \*arg1, \*arg2;**

Модуль трех операндов:

\*dest = \*arg1 % \*arg2;

Bmodulo предполагает, что \*arg2 является положительным

**double \***

**Bxtofy(dest, arg1, arg2) double \*dest, \*arg1, \*arg2;**

Возведение в степень трех операндов:

\*arg 2

\*dest = \*arg1

**Группа** **2**

**double \***

**s2Bcd(ddest, strsourse) double \*ddest; char \*strsourse;**

строка в тип double:

ddest - это адрес переменной double, в которую помещается значение числа в строке strsource.

**char \***

**Bcd2s(strdest, dsource) char \*strdest; double \*dsource;**

тип double в строку:

Эта функция является обратной для s2bcd, где strdest должно быть достаточно длинным, чтобы соответствовать представлению данного номера dsource.

Максимальный размер двойного числа в настоящее время составляет двадцать один байт. Это включает показатель степени и два знака, а также дополнительный байт для нулевого завершения.

**double \***

**Bneg(dest, arg1) double \*dest, \*arg1;**

Отрицание двух операндов:

\*dest = -\*arg1

**double \***

**BUneg(dest) double \*dest;**

Отрицание одного операнда:

\*dest = -\*dest

**double \***

**Babs(dest, arg1) double \*dest, \*arg1;**

Абсолютное значение двух операндов:

\*dest = |\*arg1|

**double \***

**BUabs(dest) double \*dest;**

Абсолютное значение одного операнда:

\*dest = |\*dest|

**Bint(dest, arg1) double \*dest, \*arg1;**

Возвращает целочисленную часть \*arg1 в \*dest (в направлении 0).

**Bentier(dest, arg1) double \*dest, \*arg1;**

Возвращает нижнюю целую часть числа \*arg1 в \*dest (в направлении бесконечности)

**double \***

**Blog(dest, arg1) double \*dest, \*arg1;**

Возвращает натуральный логарифм \*arg1 в \*dest.

\*dest = log(\*arg1);

**double \***

**Bsqr(dest, arg1) double \*dest, \*arg1;**

Возвращает квадратный корень из \*arg1 в \*dest.

\*dest = sqrt(\*arg1)

\*arg1 проверяется на отрицательный результат.

**double \***

**Bexp(dest, arg1) double \*dest, \*arg1;**

Возводит e (e = 2.7...) в степень \*arg1:

\*arg1

\*dest = e

\*arg1 проверяется на предмет слишком большого размера.

**double \***

**Bfac(dest, arg1) double \*dest, #arg1;**

Принимает факториал \*arg1:

\*dest = (\*arg1)!;

\*arg1 проверяется на предмет отрицательного или слишком большого размера.

**Трансцендентные функции:**

**double \***

**Bsin(dest, arg1) double \*dest, \*arg1;**

Возвращает синус \*arg1 в \*dest:

\*dest = sin(\*arg1)?

**double \***

**Bcos(destr arg1) double \*dest, \*arg1;**

Возвращает косинус \*arg1 в \*dest:

\*dest = cos(\*arg1);

**double \***

**Btan(dest, arg1) double \*dest, \*arg1;**

Возвращает тангенс \*arg1 в \*dest:

\*dest = tan(\*arg1);

**double \***

**Barctan(dest, arg1) double \*dest, \*arg1;**

Устанавливает \*dest равным арктангенсу \*arg1:

\*dest = arctan(\*arg1);

**Группа 3**

**Bsign(x) double \*x;**

Возвращает 1, если \*x положительно;

0, если \*x равно 0;

-1, если \*x иное.

**Blt(x, y) double \*x, \*y;**

Возвращает 1, если \*x < \*y

**Bgt(x, y) double \*x, \*y;**

Возвращает 1, если \*x > \*y

Beq(x, y) double \*x, \*y;

Возвращает 1, если \*x == \*y

**Bne(x, y) double \*x, \*y;**

Возвращает 1, если \*x != \*v

**Ble(x, y) double \*x, \*y;**

Возвращает 1, если \*x <= \*y

**Bge(x, y) double \*x, \*y;**

Возвращает 1, если \*x >= \*y

**Btest(x, y) double \*x, \*y**

Возвращает 1, если \*x > \*y;

0 if \*x == \*y;

-1 если иное.

**Bseterrflg(e)**

Устанавливает глобальный Berrflg в значение e, если он очищен.

**Группа 4**

**double \***

**Bmant(dest, n) double \*n; char \*dest;**

Помещает мантиссу \*n в \*dest.

Функции Bmant и Bexpo возвращают значения a и b соответственно, где аргумент x имеет вид x = a\*10\*\*b для .1 <= |a| < 1.

(кроме случаев, когда x == 0).

**Bexpo(x) double \*x;**

Возвращает целочисленный показатель степени \*x.

Функции Bmant и Bexpo возвращают значения a и b соответственно, где аргумент x имеет вид x = a\*10\*\*b для .1 <= |a| < 1.

(кроме случаев, когда x == 0).

**Bmov(dest, arg1) double \*dest, \*argi;**

Копирует \*arg1 и \*dest:

\*dest = \*arg1;

**Bround(dest, i) double \*dest; int i;**

Округляет \*dest, добавляя 5.0Ei к \*dest (мантисса 5 с показателем i).

**int2bcd(dest, i) double \*dest; int i;**

Преобразует целое в число удвоенной точности.

**Bcd2int(arg1) double \*arg1;**

Возвращает целую часть \*arg1, округленную от нуля.

**double \***

**Bcheby(res, x, coef, n) double \*res, \*x, coef[n][BCDS]; int**

Возвращает в res n-е приближение функции, коэффициенты Чебышева которой находятся в coef, вычисленном как \*x.

# Приложение H Функции длинных целых

Следующие функции доступны в файле LONG.C.

**cclprt(a) long \*a;**

Печать \*a шестнадцатеричной системе счисления.

**cclneg(d) long \*d;**

Один операнд отрицание:

\*d = -\*d;

**cclinc(a) long \*a;**

Приращение одного операнда:

++\*a

**cc1dec(a) long \*a;**

Декремент одного операнда:

--\*a

**cc1com(a) long \*a;**

Дополнение к одному операнду:

\*a = ^\*a

**cc2mov(a, b) long \*a, \*b;**

Копия двух операндов:

\*a = \*b

**cc2add(a, b) long \*a, \*b;**

Сложение двух операндов:

\*a += \*b

**cc2com(a, b) long \*a, \*b;**

Дополнение двух операндов:

\*a = ~\*b

**cc2or(a, b) long \*a, \*b;**

Два операнда логическое или:

\*a |= \*b

**cc2xor(a, b) long \* a, \*b;**

Два операнда xor:

\*a ^= \*b

**cc2cmp(a, b) long \*a, \*b;**

Сравнение двух операндов:

Возвращает 1 если \*x > \*y;

0 если \*x == \*y;

-1 если иначе.

**cc2mul(pa, pb) long \*pa, \*pb;**

Умножение двух операндов:

\*pa \*= \*pb;

**cc2imul (x, b0, b1) long \*x; int b0, b1;**

Два операнда немедленного умножения, где b0, b1-целые числа, образующие наиболее значимую и наименее значимую часть (соответственно) непосредственного длинного:

\*x \*= (b0, b1)

**cc2iadd(x, b0, b1) long \*x; int b0, b1;**

Два операнда немедленного сложения, где b0, b1-целые числа, образующие наиболее значимую и наименее значимую часть (соответственно) непосредственного длинного:

\*x += (b0, b1)

**cc2and(a, b) long \*a, \*b;**

Два операнда логические и:

\*a &= \*b

**cc2neg(a, b) long \*a, \*b;**

Отрицание двух операндов:

\*a = -\*b

**cc2sub(a, b) long \*a, \*b;**

Вычитание двух операндов:

a -= \*b

**cc3add(q, a, b) long q, \*a, \*b;**

Сложение трех операндов:

\*q \* \*a + \*b

**cc3and(a, b, c) long \*a, \*b, \*c;**

Три операнда логические и:

\*a \* \*b & \*c

**cc3or(a, b, c) long \*a, \*b, \*c;**

Три операнда логического ИЛИ:

\*a = \*b | \*c

**cc3xor(a, b, c) long \*a, \*b, \*c;**

Три операнда исключающее ИЛИ:

\*a = \*b ^ \*c

**cc3sub(a, b, c) long \*a, \*b, \*c;**

Вычитание трех операндов:

\*a = \*b - \*c

**ccldiv(d, s, ccxdrem) long \*d, \*s, \*ccxdrem;**

Форма трех операндного подписанного деления с остатком:

\*d /= \*s

\*ccxdrem = \*d % \*s