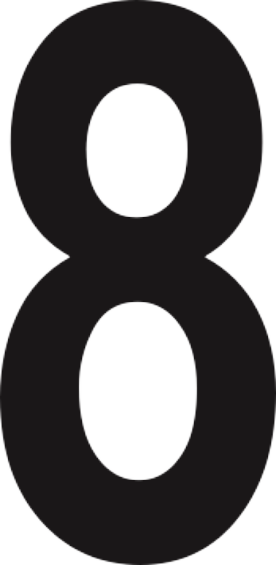
****

**Кросс-компилятор С для**

**микроконтроллеров Z80 и Z180**



HI**-TECH Software**

A division of Gretetoy Pty. Ltd. ACN 002 724 549

PO Box 103 Alderley QLD 4051 Australia

Phone +61 7 3355 8333 Fax +61 7 3355 8334

E-mail: [hitech@htsoft.com](mailto:hitech@htsoft.com)

Web: <http://www.htsoft.com>

FTP: [ftp.htsoft.com](ftp://ftp.htsoft.com)

Вторая печать (c), март 2000

Авторское право © 1984-2000 HI-TECH Software

Все права защищены

Распечатано в Австралии[[1]](#footnote-1)

**ВЫ ДОЛЖНЫ ВНИМАТЕЛЬНО ПРОЧИТАТЬ СЛЕДУЮЩЕЕ ПЕРЕД УСТАНОВКОЙ ИЛИ ИСПОЛЬЗОВАНИЕМ ДАННОГО ПРОГРАММНОГО ПАКЕТА. ЕСЛИ ВЫ НЕ СОГЛАСНЫ С УКАЗАННЫМИ НИЖЕ УСЛОВИЯМИ, ВЫ ДОЛЖНЫ НЕМЕДЛЕННО ВЕРНУТЬ ВЕСЬ ПАКЕТ ВАШЕМУ ПОСТАВЩИКУ И ВАШИ ДЕНЬГИ БУДУТ ВОЗВРАЩЕНЫ. ИСПОЛЬЗОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ОЗНАЧАЕТ ВАШЕ СОГЛАСИЕ С ЭТИМИ УСЛОВИЯМИ**

Чтобы убедиться, что вы получите преимущества гарантии, описанной ниже, вы должны заполнить и подписать предлагающуюся регистрационную карточку и сразу вернуть ее в HI-TECH Software.

**Лицензионное соглашение**

HI-TECH Software, a division of Gretetoy Pty. Ltd., of 12 Blackwood St. Mitchelton QLD 4053 Australia, предоставляет этот пакет программного обеспечения для использования на следующих условиях:

Этот пакет программного обеспечения полностью защищен авторским правом HI-TECH Software и в любых случаях остается собственностью HI-TECH Software.

**Вы можете:**

* Использовать этот пакет программного обеспечения на одном компьютере. Вы можете переносить этот пакет с одной компьютерной системы на другую при условии, что вы используете его только на одном компьютере одновременно.
* Делать копии дискет с предоставленным пакетом программного обеспечения в резервных целях, если все копии помечены именем пакета программного обеспечения и содержат уведомление об авторском праве HI-TECH Software.
* Использовать пакет программного обеспечения для создания собственных программ. Если такие программы не содержат какой-либо части этого программного пакета кроме извлечений из любых включенных объектных библиотек, то эти программы останутся вашей собственностью и не подпадают под действие данного соглашения.
* Передавать пакет программного обеспечения и эту лицензию третьей стороне при условии, что третья сторона соглашается с условиями этой лицензии, и что все копии пакета программного обеспечения передаются третьей стороне или уничтожены. Третья сторона должны сообщить HI-TECH Software о своем согласии с условиями этой лицензии.

**Вы не можете:**

* Продавать, одалживать, дарить или передавать копии программного пакета любым другим физическим или юридическим лицам, за исключением условий указанных выше, и не позволять другим людям делать копии этого программного пакета.
* Включать любую часть этого пакета программного обеспечения в свои собственные программы, за исключением объединения в исполняемой форме только извлечений из любых объектных библиотек.
* Использовать этот пакет для разработки приложений жизнеобеспечения или любого приложения, где отказ приложения может привести к смерти или телесным повреждениям любому человеку. Если вы используете это программное обеспечение, для разработки любого такого приложения, вы соглашаетесь взять на себя всю ответственность за любые такие отказы и обязуетесь оградить HI-TECH Software в отношении любых и всех претензий, вытекающих из любых подобных сбоев.

**Период действия**

Данная лицензия действует до момента его расторжения. Вы можете расторгнуть ее, вернув HI-TECH Software или уничтожив все копии программного пакета. Она также прекращается, если вы не соблюдаете любое из вышеуказанных условий.

**Гарантии**

HI-TECH Software гарантирует, что имеет право предоставить вам эту лицензию и, что пакет программного обеспечения не является объектом авторского права, на которое HI-TECH Software не имеет прав. В некоторых штатах и федеральных законах могут быть предусмотрены дополнительные гарантии.

**Ограничение ответственности**

Этот программный пакет поставляется с честными намерениями и считается самого высокого качества. Из-за природы процесса разработки программного обеспечения, вполне возможно, что есть скрытые дефекты программного обеспечения, которые могут повлиять на его использование или функционирование любого программного обеспечения или устройства, разработанного с помощью этого пакета. Вы принимаете на себя всю ответственность за определение, является ли этот пакет подходящим для вашего приложения, и для обеспечения корректной работы в прикладном программном обеспечении и оборудовании. Единственная и максимальная ответственность HI-TECH Software за любые дефекты в этом пакете ограничена суммой, которую вы заплатили за лицензию на использование этого программного обеспечения. Ни при каких обстоятельствах HI-TECH Software не несет ответственности за любые косвенные убытки, если такое исключение запрещено по закону.

**Товарные знаки**

Ниже представлены товарные знаки HI-TECH Software:

Pacific C; HI-TECH C; Lucifer; PPD; HPD

Другие товарные знаки и зарегистрированные товарные знаки, используемые в этом документе, являются собственностью их соответствующих владельцев.

Техническая поддержка

Для технической поддержки компилятора HI-TECH C, вы должны связаться с HI-TECH Software одним из способов, перечисленных в нижеследующей таблице. Для получения технической поддержки необходимо зарегистрировать свой компилятор, отправив регистрационную карточку найденную внутри передней обложки настоящего руководства. Это дает вам 3 месяца бесплатной технической поддержки с момента, когда вы сделаете свой первый запрос в службу технической поддержки. Вы также будете иметь право на одно бесплатное обновление, которое будет выслано вам автоматически, если вы вернули вашу регистрационную карточку.

После начального периода бесплатного сопровождения вы можете заключить годовое соглашение о поддержке этого компилятора. Свяжитесь с HI-TECH Software или вашим поставщиком для получения информации о ценах. Соглашение о годовом сопровождении предоставит вам приоритетный доступ к технической поддержке и всем обновлениям, отправляемым вам автоматически.

Если вы не хотите оплачивать ежегодное сопровождение, вы можете отправлять запросы в техническую поддержку, но на них будут отвечать только по мере возможности. Это может привести к значительным задержкам. Небольшие обновления можно получить бесплатно, загрузив файлы патчей с наших WWW и ftp-серверов. Они будут исправлять проблемы с выпущенными версиями компилятора. Они не будут обновлять с одной основной версии на другую. Обновления можно приобрести отдельно.

|  |  |
| --- | --- |
| Всемирная сеть | <http://www.htsoft.com> |
| Электронная почта | [support@htsoft.com](mailto:support@htsoft.com) |
| Факс | +61 7 3355 8333 |
| Телефон | +61 7 3355 8334 |
| Почта | PO Box 103 ALDERLEY  QLD 4051 Australia |

Также вы можете обратиться к следующим дистрибьюторам в других странах за консультациями по поддержке и продажам - **за технической поддержкой сначала всегда обращайтесь к своему поставщику**. Политика отдельных торговых посредников может изменяться в зависимости от бесплатной поддержки.

**Содержание**

1 Введение 1

1.1 Условные обозначения 1

1.2 Кросс-компилятор HI-TECH C Z80 1

1.3 Установка 1

1.3.1 MS-DOS 1

1.3.2 Программа INSTALL 1

1.3.3 Доступ к компилятору 3

1.4 Установка в Unix 3

1.4.1 Доступ к компилятору в Unix 4

1.5 Начало работы 4

1.6 Пример программы 4

1.6.1 Распределение памяти 5

1.7 Использование HPDZ 5

1.8 Использование ZC 6

1.8.1 Выбор формата выходного файла 8

1.9 Выполнение вашей программы 8

2 Учебные руководства 9

2.1 Обзор процесса компиляции 9

2.1.1 Компиляция 9

2.1.2 Ввод компилятора 10

2.2 Секции psect и компоновщик 21

2.2.1 Секции psect 21

2.3 Соединение psect 24

2.3.1 Группировка psect 24

2.3.2 Расположение psect 24

2.3.3 Опции компоновщика расположения psect 25

2.3.4 Проблемы при компоновке 32

2.3.5 Изменение параметров компоновщика 34

3 Использование HPDZ 37

3.1 Введение 37

3.1.1 Запуск HPDZ 37

3.2 Оконный интерфейс пользователя HI-TECH 38

3.2.1 Требования к оборудованию 38

3.2.2 Цвета 38

3.2.3 Выпадающее меню 40

3.2.4 Выбор окон 41

3.2.5 Перемещение и изменение размеров окон 43

3.2.6 Кнопки 44

3.2.7 Меню настроек 44

3.3 Учебник: создание и компиляция программы на C 45

3.4 Редактор HPDZ 50

3.4.1 Рамка 50

3.4.2 Область содержимого 50

3.4.3 Строка состояния 50

3.4.4 Клавиатурные команды 51

3.4.5 Блочные команды 52

3.4.6 Редактирование буфера обмена 54

3.5 Меню HPDZ 56

3.5.1 Меню «» (Система) 56

3.5.2 Меню File (Файл) 57

3.5.3 Меню Edit (Правка) 57

3.5.4 Меню Options (Опции) 60

3.5.5 Меню Compile (Компилировать) 63

3.5.6 Меню Make (Создать) 65

3.5.7 Меню Run (Выполнить) 68

3.5.8 Меню Utility (Утилиты) 70

3.5.9 Меню Help (Справка) 71

4 Драйвер компилятора командной строки ZC 74

4.1.1 Длинные командные строки 74

4.1.2 Библиотеки по умолчанию 76

4.1.3 Стандартный стартовый (Startoff) модуль запуска времени выполнения 76

4.2 Опции компилятора ZC 76

4.2.1 -180: генерация кода Z180/64180 77

4.2.2 -64180: генерация кода Z180/64180 77

4.2.3 -Aspec: определение адресов ПЗУ и ОЗУ 77

4.2.4 -AAHEX: Генерировать файл American Automation symbolic HEX 78

4.2.5 -ALTREG: Использовать альтернативный набор регистров 78

4.2.6 -ASMLIST: Генерировать файл ассемблера .LST 78

4.2.7 -AV: Выбрать формат таблицы символов Avocet 79

4.2.8 -AVSIM: Выбрать формат таблицы символов Avocet 79

4.2.9 -BIN: Генерировать двоичный выходной файл 79

4.2.10 -Bs: Выбор малой модели памяти 79

4.2.11 -Bl: Выбор большой модели памяти 79

4.2.12 -Bc: Выбор модели памяти CP/M 80

4.2.13 -C: Компилировать в объектные файлы 80

4.2.14 -CLIST: Генерировать листинг исходного файла C 81

4.2.15 -CPM: Генерировать исполняемый файл CP/M 81

4.2.16 -CR*file*: Генерировать листинг перекрестных ссылок 81

4.2.17 -D*macro*: Определить макрос препроцессора 81

4.2.18 -E: Использовать формат "редактора" для ошибок компилятора 82

4.2.19 -E*file*: Перенаправить ошибки компилятора в файл 82

4.2.20 -G*file*: Генерировать таблицу символов уровня источника 83

4.2.21 -H*file*: Генерировать файл символов уровня ассемблера 83

4.2.22 -HELP: Отображение справки 83

4.2.23 -I*path*: Путь поиска включаемых файлов Include 84

4.2.24 -L*library*: Просматриваемая библиотека 84

4.2.25 -L-*option*: Определить опцию, передаваемую компоновщику 84

4.2.26 -M*file*: Генерация файла распределения 85

4.2.27 -MOTOROLA: Генерировать выходной HEX файл в формате Motorola S1/S9 85

4.2.28 -N*length*: Определить значимую длину идентификатора 85

4.2.29 -O: Включить локальную оптимизацию 85

4.2.30 -O*file*: Определить имя выходного файла 85

4.2.31 -OF: Оптимизация по скорости 86

4.2.32 -OMF51: Произвести выходной файл OMF-51 86

4.2.33 -P: Предварительная обработка ассемблерных файлов 86

4.2.34 -P8: Использовать 8 разрядную адресацию портов 86

4.2.35 -P16: Использовать 16 разрядную адресацию портов 86

4.2.36 -PRE: Создание исходного кода обработанного препроцессором 86

4.2.37 -PROTO: Сгенерировать прототипы функций 86

4.2.38 -PSECTMAP: Отобразить полное использование памяти 87

4.2.39 -q: Тихий режим 87

4.2.40 –ROMDATA: Оставить данные инициализации в ПЗУ 88

4.2.41 –ROMranges: Определите диапазоны ПЗУ для кода 88

4.2.42 -S: Компиляция в ассемблерный код 88

4.2.43 -SA: Компиляция в исходные файлы ассемблера AVMAC Avocet 88

4.2.44 -STRICT: Включить строгое соответствие ANSI 88

4.2.45 -TEK: Генерировать выходной HEX файл в формате Tektronix 89

4.2.46 -U*macro*: Отмена определения макроса 89

4.2.47 -UBROF: Генерировать выходной файл в формате UBROF 89

4.2.48 -UNSIGNED: Сделать символьный тип по умолчанию unsigned 89

4.2.49 -V: Многословная компиляция 89

4.2.50 -W*level*: Установить уровень предупреждений 89

4.2.51 -X: Исключить локальные символы 89

4.2.52 -Z180: Генерировать код Z180 90

4.2.53 -Zg: Глобальная оптимизация 90

5 Функции и окружение времени выполнения 91

5.1 Форматы выходных файлов 91

5.2 Файлы символов 92

5.2.1 Таблицы символов Avocet 92

5.3 Предопределенные макросы 92

5.4 Поддерживаемые типы данных 93

5.4.1 8 разрядные целочисленные типы данных 93

5.4.2 16 разрядные целочисленные типы данных 93

5.4.3 32 разрядные целочисленные типы данных 93

5.4.4 Числа с плавающей точкой 94

5.5 Абсолютные переменные 94

5.6 Квалификатор типа port 94

5.7 Структуры и объединения 95

5.7.1 Битовые поля в структурах 95

5.8 Квалификаторы const и volatile 96

5.9 Специальные квалификаторы типа 96

5.9.1 Квалификатор типа persistent 97

5.9.2 Квалификатор типа code 97

5.10 Указатели 97

5.10.1 Совмещение квалификаторов типа и указателей 97

5.10.2 Указатели code 98

5.10.3 Указатели const 98

5.11 Обработка прерываний в C 99

5.11.1 Макросы обработки прерываний 100

5.11.2 Режимы прерываний 102

5.11.3 Предопределенные имена векторов прерываний 105

5.11.4 Обработка немаскируемых прерываний 105

5.11.5 Быстрые прерывания 105

5.12 Смешивание кода C и ассемблера Z80 106

5.12.1 Внешние функции на языке ассемблера 106

5.12.2 #asm, #endasm и asm() 106

5.13 Проверка сигнатуры 107

5.14 Компоновка программ 108

5.15 Использование памяти 109

5.16 Использование регистров 109

5.17 Организация стекового кадра 109

5.18 Передача аргумента функции 110

5.19 Значения возвращаемые функцией 112

5.19.1 Возвращение 8 разрядных значений 112

5.19.2 Возвращение 16 разрядных значений 112

5.19.3 Возвращение 32 разрядных значений 112

5.19.4 Возвращение значений структуры 112

5.20 Соглашения о вызовах функций в большой модели 113

5.20.1 Функции near и basenear в большой модели 114

5.21 Стек и выделение кучи 115

5.22 Локальные переменные 115

5.22.1 Переменные auto 115

5.22.2 Статические переменные 115

5.23 Секции psect генерируемые компилятором 115

5.24 Модули начальной загрузки времени выполнения 117

5.24.1 Процедура powerup 118

5.24.2 Использование символов определенных компоновщиком 120

5.24.3 Настройка кода Startoff времени выполнения 120

5.25 Оптимизация кода Z80 121

5.26 Директивы pragma 121

5.27 Стандартные функции ввода-вывода и последовательный ввод-вывод 124

6 Макроассемблер Z80 126

6.1 Использование ассемблера 126

6.2 Опции ассемблера 127

6.3 Язык ассемблера Z80 128

6.3.1 Набор символов 128

6.3.2 Константы 128

6.3.3 Разделители 129

6.3.4 Специальные символы 129

6.3.5 Идентификаторы 129

6.3.6 Строки 131

6.3.7 Выражения 131

6.3.8 Формат оператора 132

6.3.9 Секции программы 132

6.3.10 Расширенные коды условий 133

6.3.11 Директивы ассемблера 133

6.3.12 Макро-вызовы 141

6.3.13 Элементы управления ассемблером 141

7 Справочник по компоновщику и утилитам 143

7.1 Введение 143

7.2 Перемещение и секции psect 143

7.3 Секции программ 143

7.4 Локальные psect 144

7.5 Глобальные символы 144

7.6 Адреса ссылки и загрузки 144

7.7 Функционирование 144

7.7.1 Числа в опциях компоновщика 146

7.7.2 -A*class=low-high,...* 146

7.7.3 -C*x* 146

7.7.4 -C*psect=class* 146

7.7.5 -D*class=delta* 146

7.7.6 -D*symfile* 147

7.7.7 -E*errfile* 147

7.7.8 -F 147

7.7.9 -G*spec* 147

7.7.10 -H*symfile* 148

7.7.11 -H+*symfile* 148

7.7.12 -J*errcount* 148

7.7.13 -K 148

7.7.14 -I 148

7.7.15 -L 148

7.7.16 -LM 148

7.7.17 -M*mapfile* 149

7.7.18 –N, -Ns и  -Nc 149

7.7.19 -O*outfile* 149

7.7.20 -P*spec* 149

7.7.21 -Q*processor* 150

7.7.22 -S 150

7.7.23 -S*class=limit[,bound]* 150

7.7.24 -U*symbol* 151

7.7.25 -V*avmap* 151

7.7.26 -W*num* 151

7.7.27 -X 151

7.7.28 -Z 151

7.8 Вызов компоновщика 151

7.9 Карты распределения памяти 152

7.9.1 Информация о графе вызовов 152

7.10 Библиотеки 154

7.10.1 Формат библиотеки 155

7.10.2 Использование библиотек 155

7.10.3 Примеры 156

7.10.4 Предоставление аргументов 156

7.10.5 Формат листинга 156

7.10.6 Упорядочивание библиотек 157

7.10.7 Сообщения об ошибках 157

7.11 OBJTOHEX 157

7.11.1 Спецификация контрольной суммы 157

7.12 CREF 158

7.12.1 -F*prefix* 158

7.12.2 -H*heading* 159

7.12.3 -L*len* 159

7.12.4 -O*outfile* 159

7.12.5 -P*width* 159

7.12.6 -S*stoplist* 159

7.12.7 -X*prefix* 159

7.13 MEMMAP 160

7.13.1 Использование MEMMAP 160

8 Отладчик уровня исходного кода Lucifer 161

8.1 Использование Lucifer 161

8.2 Имена символов в выражениях 162

8.2.1 Автоматические переменные и параметры 163

8.3 Список команд Lucifer 163

8.3.1 Команда B: Установка или отображение точек останова 163

8.3.2 Команда C: Отображение инструкции в PC 164

8.3.3 Команда D: Отображение содержимого памяти 164

8.3.4 Команда E: Исследование исходного кода C 164

8.3.5 Команда G: Начало выполнения 165

8.3.6 Команда I: Переключение режима трассировки инструкций 165

8.3.7 Команда L: Загрузка шестнадцатеричного файла 165

8.3.8 Команда M: Изменение памяти 165

8.3.9 Команда P: Переключение приглашения ввода 165

8.3.10 Команда Q: Выход в операционную систему 166

8.3.11 Команда R: Удаление контрольной точки 166

8.3.12 Команда S: Шаг на одну строку 166

8.3.13 Команда T: Трассировка одной инструкции 167

8.3.14 Команда U: Дизассемблирование машинных команд 167

8.3.15 Команда W: Загрузка двоичного файла 167

8.3.16 Команда X: Исследование или изменение регистров 167

8.3.17 Команда @: Отображение переменных C 168

8.3.18 Команда .: Установка контрольной точки и переход 168

8.3.19 Команда ;: Отображение от исходной строки 169

8.3.20 Команда =: Отображение следующей страницы источника 169

8.3.21 Команда -: Отображение предыдущей страницы источника 169

8.3.22 Команда /: Поиск строки в исходном файле 169

8.3.23 Команда !: Выполнение команды DOS 170

8.3.24 Другие команды 170

8.4 Ввод вывод пользователя в Lucifer 170

8.5 Установка Lucifer на целевой компьютер 170

8.5.1 Изменение целевого кода 170

9 Сообщения об ошибках 171

10 Библиотечные функции 234

11 Предметный указатель 324

**Таблицы**

Таблица 2‑1 Конфигурационные файлы 9

Таблица 2‑2 Типы входных файлов 12

Таблица 2‑3 Вывод листинга 13

Таблица 2‑4 Вывод препроцессора 14

Таблица 2‑5 Промежуточные и вспомогательные файлы 14

Таблица 2‑6 Вывод анализатора 15

Таблица 2‑7 Вывод генератора кода 16

Таблица 2‑8 Вывод ассемблера 18

Таблица 2‑9 Листинг ассемблера 19

Таблица 2‑10 Выходные форматы 21

Таблица 3‑1 Значения цветов 39

Таблица 3‑2 Цвета атрибутов 39

Таблица 3‑3 Настройки цветов кода 40

Таблица 3‑4 Клавиши и действия мыши системного меню 40

Таблица 3‑5 Горячие клавиши вызова меню HPDZ 42

Таблица 3‑6 Клавиши режима изменения размера 43

Таблица 3‑7 Клавиши операций с блоками 52

Таблица 3‑8 Клавиши редактирования 53

Таблица 3‑9 Макросы, используемые в командах пользователей 71

Таблица 4‑1 Типы файлов ZC 74

Таблица 4‑2 Опции ZC 75

Таблица 4‑3 Библиотеки малой модели памяти 79

Таблица 4‑4 Библиотеки большой модели памяти 80

Таблица 4‑5 Библиотеки модели памяти CP/M 80

Таблица 5‑1 Форматы выходных файлов 91

Таблица 5‑2 Предопределенные символы CPP 92

Таблица 5‑3 Формат 32 разрядного числа с плавающей точкой 94

Таблица 5‑4 Макросы и функции поддержки прерываний 99

Таблица 5‑5 Вектора прерываний Z180 105

Таблица 5‑6 Стандартные модули startoff времени выполнения 117

Таблица 5‑7 Действия powerup по умолчанию 120

Таблица 5‑8 Директивы pragma 122

Таблица 5‑9 Поддерживаемые функции STDIO 124

Таблица 6‑1 Опции ассемблера ZAS 126

Таблица 6‑2 Числа и основания ZAS 128

Таблица 6‑3 Операторы 131

Таблица 6‑4 Формат операторов ZAS 132

Таблица 6‑5 Расширенные коды условий 133

Таблица 6‑6 Директивы ZAS (псевдооперации) 134

Таблица 6‑7 Флаги PSECT 135

Таблица 6‑8 Управление ассемблером ZAS 141

Таблица 7‑1 Опции компоновщика 145

Таблица 7‑2 Опции библиотекаря 155

Таблица 7‑3 Ключевые буквы команд библиотекаря 155

Таблица 7‑4 Опции OBJTOHEX 158

Таблица 7‑5 Опции CREF 159

Таблица 7‑6 Опции MEMMAP 160

Таблица 8‑1 Формы выражения Lucifer 162

Таблица 8‑2 Список команд Lucifer 163

Таблица 8‑3 Варианты команд @ Lucifer 168

**Рисунки**

Рисунок 2‑1 Обзор компиляции 11

Рисунок 3‑1 Экранная заставка HPDZ 37

Рисунок 3‑2 Диалог настройки 45

Рисунок 3‑3 Программа Hello в HPDZ 46

Рисунок 3‑4 Меню File HPDZ 47

Рисунок 3‑5 Диалоговое окно адреса ПЗУ и ОЗУ 48

Рисунок 3‑6 Окно ошибок 49

Рисунок 3‑7 Меню Edit HPDZ 58

Рисунок 3‑8 Меню Options HPDZ 60

Рисунок 3‑9 Меню Compile HPDZ 64

Рисунок 3‑10 Меню Make HPDZ 65

Рисунок 3‑11 Меню Options HPDZ 69

Рисунок 3‑12 Меню Utility HPDZ 70

Рисунок 3‑13 Меню Help HPDZ 72

Рисунок 4‑1 Префиксы и суффиксы библиотеки 84

Рисунок 5‑1 Кадр стека после входа в функцию 109

# Введение

## Условные обозначения

В этом руководстве принято соглашение, что любой текст, который нужно ввести, напечатан **жирным** **шрифтом**. Компьютерные подсказки и ответы печатаются в формате с постоянным интервалом, который выделяется жирным шрифтом там, где вам необходимо его ввести. Особенно полезные места и новые термины выделены *курсивом*. Для программы с оконным интерфейсом, такой как HPD, некоторые понятия трудно передать в тексте. Они представлены с помощью коротких обучающих программ и демонстрационных экранов.

## Кросс-компилятор HI-TECH C Z80

В этом руководстве рассматривается кросс-компилятор HI-TECH C Z80 для семейства микроконтроллеров Z80. В этом руководстве представлена информация об установке, использовании и настройке компилятора.

Компилятор работает под управлением MS-DOS, Unix и Xenix. Для использования под MS-DOS HI-TECH C требует процессор 8088, 8086, 80186, 80286, 80386, 80486 или Pentium, имеющий по меньшей мере 512 Кбайт свободной памяти и жесткий диск. Требуется MS-DOS 3.1 или более поздняя версия. Мы рекомендуем MS-DOS 3.3 или более позднюю версию, или DR-DOS 6.0 или более позднюю версию. Мы настоятельно рекомендуем иметь по крайней мере 1 Мбайт свободной памяти XMS (от HIMEM.SYS). Мышь не требуется, но настоятельно рекомендуется.

## Установка

Процесс установки зависит от используемой вами операционной системы.

### MS-DOS

Кросс-компилятор HI-TECH C Z80 поставляется на двух или более 3.5" или 5.25" дискетах. Содержание дисков перечислено в файле с названием PACKING.LST на диске 1. Для установки компилятора, необходимо использовать программу INSTALL на диске 1. Она расположена в корневом каталоге диска.

Поместите диск 1 в дисковод для гибких дисков, затем введите a:install или b:install при необходимости. Затем программа INSTALL отобразит ряд сообщений и при необходимости запросит различную информацию. Возможно необходимо проверить переменные окружения (используя команду SET), в случае, если имеется установленная переменная окружения под названием TEMP. Если эта переменная установлена, ее значение должно быть путем к каталогу. Полный путь должен существовать и определять каталог, в котором могут быть созданы временные файлы. Если переменная TEMP определяет несуществующий каталог, программа INSTALL может завершиться ошибкой.

### Программа INSTALL

Программа INSTALL использует несколько экранных окон. Есть окно *сообщений*, в котором на экране отображаются сообщения и подсказки. Есть также одно-строчное окно *состояния*, в котором INSTALL показывает, что делает в настоящее время. Другие окна будут всплывать время от времени.

Диалоговое или *аварийное* окно раскроется, когда INSTALL потребуется ваше действие, или при возникновении любой ошибки. Они предлагают возможность продолжить (Continue), повторить (Retry), при необходимости, или прекратить (Terminate). Если вы выберете TERMINATE (прекратить), установка будет неполной. Перемещаемый ползунок ориентировочно указывает степень завершения установки.

#### Шаги установки

Когда INSTALL запросит действие, вы можете: нажать Enter для продолжения, ESCAPE для завершения или использовать мышь, если она у вас есть, для выбора кнопок, отображаемых в диалоговом окне. Чтобы использовать мышь, переместите курсор на нужную кнопку, а затем нажмите и отпустите левую кнопку мыши.

Сначала INSTALL просто сообщит, что собирается установить пакет программного обеспечения HI-TECH. Выберите CONTINUE или нажмите Enter. Вам будет предложено выбрать полную установку без вопросов или пользовательскую установку. Пользовательская установка позволяет не устанавливать дополнительные компоненты компилятора. Пользовательская установка также позволит указать каталоги, в которые будет установлен компилятор.

#### Пользовательская установка

При выборе пользовательской установки, INSTALL задаст ряд вопросов о путях к каталогу для установки компилятора. В каждом из них отображается путь по умолчанию и нужно нажать Enter для подтверждения этого пути или ввести новый путь, а затем нажать Enter. Вы можете выбрать TERMINATE, если не хотите продолжать. Обратите внимание, что INSTALL создаст любой необходимый каталог. Однако она не создает промежуточные каталоги, например, она создаст каталог C:\COMPILE\HITECH, если он не существует, но в этом случае она не создаст каталог COMPILE. Он уже должен существовать.

INSTALL также запрашивает каталог для временных файлов. Это каталог, в котором компилятор будет размещать временные файлы. Им может быть RAM-диск, если он существует (но убедитесь, что RAM-диск достаточно большой - не менее нескольких сотен килобайт), или им может быть каталог на жестком диске или просто оставьте поле незаполненным. Если оно будет оставлено незаполненным, то компилятор будет создавать временные файлы в рабочем каталоге.

Далее INSTALL задаст несколько вопросов об установке дополнительных частей компилятора. Для каждой части вы можете ответить Да (Enter) или Нет (F10) или использовать мышь, чтобы нажать соответствующую кнопку.

#### Серийный номер и ключ установки

После ответов на эти вопросы, или немедленно, если вы выбрали полную установку, INSTALL попросит вас ввести серийный номер и ключ установки. Также вам будет предложено ввести свое имя и название вашей компании.

INSTALL свяжет установленный компилятор с этой информацией. Серийный номер и ключ установки находятся на обратной стороне титульной страницы руководства. Серийный номер и ключ должны быть введены правильно, иначе установка будет прервана.

После этого INSTALL приступает к копированию файлов, содержащихся в основном компиляторе и дополнительных частях. Сжатые файлы распаковываются автоматически. Каждый скопированный файл будет отображен в окне состояния. Если INSTALL обнаружит, что копирует файл который уже существует, т.е. она собирается перезаписать файл, появится диалоговое окно, спрашивающее, хотите ли вы перезаписать файл. Если вы отвечаете Yes (Да) в первый раз, когда это происходит, вас спросят, хотите ли вы получать запросы о перезаписи других файлов. Отрицательный ответ заставит установку молча перезаписывать любые существующие файлы. Это допустимо, при выполнении переустановки или обновлении версии.

Во время процесса копирования INSTALL может в середине экрана отобразить окно, содержащее информативный текст из файла на дистрибутивном диске. Оно будет содержать информацию о компиляторе и других пакетах программного обеспечения HI-TECH. При чтении этой информации вы можете использовать клавиши со стрелками вверх и вниз, и клавиши Page-up и Page-down для прокрутки текста.

Программа INSTALL вызовет диалоговое окно всякий раз, когда вам нужно будет заменить диски. Когда это произойдет, поместите требуемый диск в дисковод для гибких дисков и нажмите ENTER. По завершении установки она, при необходимости, отредактирует файлы AUTOEXEC.BAT и CONFIG.SYS. Когда это произойдет, она отобразит два окна редактирования, содержащих новую и старую версии файла. Вы можете прокручивать окна и сравнивать их, чтобы увидеть, какие изменения были внесены. Вы можете отредактировать новую версию, если пожелаете. Когда закончите, нажмите F1 или щелкните кнопку DONE (готово) в окне состояния. Нажатие ESC или щелчок в окне ABORT будут препятствовать INSTALL обновить файл. Этот шаг будет пропущен, если AUTOEXEC.BAT не нуждается в модификации.

После того, как INSTALL отобразит несколько сообщений, вам следует нажать Enter, чтобы прочитать примечания к выпуску для компилятора. Это приведет к загрузке файла READ.ME в окно экрана и позволит вам прочитать его. Этот файл также автоматически копируется на жесткий диск. При повторном нажатии Enter произойдет выход в DOS.

На этом этапе может потребоваться перезагрузка, чтобы изменения в файле AUTOEXEC.BAT вступили в силу. Программа INSTALL предложит выполнить ее, если это необходимо. Теперь установка завершена.

### Доступ к компилятору

Процесс установки включит в переменную среды PATH каталог, содержащий исполняемые программы компилятора. Однако возможно, что некоторые другие программы, уже установленные в вашей системе, могут иметь то же имя, что и одна из программ-компилятора. Чтобы преодолеть это, вам может потребоваться реорганизовать переменную среды PATH.

Драйверами компилятора выступают программы ZC.EXE (версия командной строки) и HPDZ.EXE (интегрированная версия). В основном только эти команды, необходимы для доступа к компилятору, но вы также можете запускать другие утилиты напрямую.

## Установка в Unix

Кросс-компилятора Z80 C версии Unix обычно предоставляются на дискетах или лентах в формате TAR. Для установки компилятора, сначала нужно создать каталог. По умолчанию драйвер компилятора будет ожидать каталог /usr/hitech, но его можно переопределить, установив переменную окружения HTC\_Z80 так же, как для MS-DOS.

Чтобы извлечь компилятор, сначала создайте каталог, в который вы будете устанавливать компилятор:

**mkdir /usr/hitech**

Измените имя каталога на выбранное. Затем перейдите в каталог:

**cd /usr/hitech**

затем извлеките файлы с дискеты или магнитной ленты. Замените имя устройства любым физическим устройством, которое вы будете использовать для чтения ленты или диска:

**tar xf /dev/install**

Теперь компилятор установлен. Вы должны добавить к своей переменной окружения PATH каталог bin, например /usr/hitech/bin. Если каталог отличается от /usr/hitech необходимо установить переменную окружения в файле .cshrc или .profile. Например, в .cshrc (для c-shell) добавьте строку:

**setenv HTC\_Z80 /home/hitech**

Пользователю оболочки Bourne shell потребуются следующие строки в .profile:

**HTC\_Z80=/home/hitech**

**export HTC\_Z80**

Снова замените имя каталога на выбранный.

### Доступ к компилятору в Unix

Компилятор Unix имеет только драйвер командной строки, HPDZ не предусмотрен. Параметры командной строки совпадают с версией для DOS - просто используйте команду ZC, как описано в последующих главах.

## Начало работы

Для новых пользователей компилятора HI-TECH C следующий раздел содержит пошаговое руководство по запуску вашей первой программы на целевой системе. Вам будет нужна рабочая Z80 или совместимая с Z80 система с ОЗУ и ПЗУ и некоторыми средствами загрузки кода или программирования СППЗУ. И, конечно, вам нужно установить компилятор, согласно описания в предыдущей главе.

Полное руководство по использованию HPDZ и ZC вы найдете в главах "[Использование HPDZ](#_Использование_HPDZ_1)" и "[Функции и среда выполнения](#_Функции_и_окружение)".

Одно должно быть ясно, при программировании встроенных устройств действительно не существует понятия "быстрый старт". Есть несколько переменных, например, аппаратные средства, память, устройства ввода-вывода и программное обеспечение и они все должны быть абсолютно правильными, иначе программа просто не будет работать. В случае сбоя встроенной программы сообщения об ошибках отсутствуют - она черный ящик. Будьте готовы все тщательно проверить, и если возможно, начните с гарантированно рабочего оборудования. Одновременная отладка аппаратного и программного обеспечения многократно усложняет решение задачи.

## Пример программы

**/\***

**\* Тестовая программа для Z180.**

**\* Мигает светодиодами, подключенными к порту ввода-вывода 0x80**

**\*/**

**static port unsigned char LED @ 0x80; /\* Определение порта LED \*/**

**main() {**

**register int i, j;**

**i = 0; /\* 0 Включает светодиоды \*/**

**for(;;) { /\* Бесконечный цикл \*/**

**LED = i;**

**for(j = 0 j<10000; j++) continue; /\* Задержка \*/**

**i = ~i;**

**}**

**}**

Пример содержит небольшую программу написанную для процессора Z80180 с 8 светодиодами, подключенными к выходному порту 0x80. Это постоянно циклично мигающие светодиоды. Если ваше оборудование отличается, а это почти наверняка так, вы должны написать аналогичную программу специально для вашего конкретного оборудования. Однако, мигающие светодиоды - это хорошее место для начала, так как они обеспечивают визуальную индикацию функции программы. Если у вас нет присоединенных светодиодов, вы можете контролировать выходную строку порта с помощью электронно-лучевого осциллографа или логического пробника. Может использоваться даже мониторинг адресных шин с помощью электронно-лучевого осциллографа или логического анализатора. Идея заключается в том, чтобы иметь возможность определить, что программа работает корректно, используя минимум ресурсов, с тем чтобы исключить из проблемы как можно больше неизвестных.

Как только первая программа запущена, с этого момента легче продвигаться, чем пытаться запускать сложную программу с самого начала. Чтобы запустить эту программу, вам нужно скомпилировать ее, используя, интегрированную среду разработки HPDZ, или драйвер компилятора командной строки ZC.

### Распределение памяти

Прежде, чем скомпилировать пример программы, вы должны знать, какую карту памяти вы используете. Если тестовое оборудование на котором вы выполните этот код, имеет стандартный Z80 или эквивалент, т.е. он имеет область памяти 64 Кб без коммутации блоков, то вы должны просто иметь блок ПЗУ по нулевому адресу и блок ОЗУ по некоторому другому адресу в первых 64 Кб. Вы должны знать начальный адрес ОЗУ и его размер.

Если вы используете Z180 или 64180, вполне вероятно, что ваша память будет располагаться по другим адресам в 512 Кб или 1 Мб памяти этих процессоров. Вероятно, ПЗУ начнется с нуля, но возможно, что ОЗУ начинается с адреса больше чем 64 Кбайт. Вам нужно будет знать этот адрес, а также выбрать адрес в пределах первых 64 Кбайт для отображения ОЗУ, например, у вас может быть ОЗУ по адресу 10000H, но вы хотите отобразить его по адресу 8000, расширяясь до FFFF. Это оставит от 0 до 7FFF для ПЗУ.

## Использование HPDZ

Чтобы ввести эту программу, просто выполните эти шаги:

* Запустите HPDZ, набрав **HPDZ**, затем нажмите **Enter**. Если вы правильно установили HPDZ, он будет в вашем пути поиска. Вы должны иметь на экране строку меню, большое окно редактирования и небольшое окно сообщения.
* Начните вводить текст программы в окне редактирования. Командные клавиши редактора позволяют использовать стандартные клавиши ПК (клавиши со стрелками и т.д.) или WordStar-совместимые нажатия клавиш.
* После ввода полной программы (с любыми изменениями, необходимыми для вашего оборудования), нажмите **ALT-S**. На экране появится сообщение с просьбой ввести имя для сохранения файла. Введите имя **SAMPLE.C** и нажмите **Return**. Файл будет сохранен на диск.
* Нажмите **ALT-P**, чтобы открыть меню опций. Используйте мышь или клавиши со стрелками, чтобы выбрать пункт **Memory** **model** **and** **chip** **type** (Модель памяти и тип микросхемы). Это откроет диалоговое окно, позволяющее вам выбрать код Z80 или Z180 и маленькую или модель с переключением банков памяти. Выберите маленькую модель, и Z80 или Z180 в зависимости от того, какой процессор вы используете. Нажмите **Return**, чтобы выйти из диалогового окна.
* Снова нажмите **ALT-P** и выберите пункт **ROM output file** (выходной файла ПЗУ). Это откроет диалоговое окно, разрешающее вам выбрать выходной формат исполняемого файла. Выберите формат файла, совместимый с вашим программатором ППЗУ. Intel HEX или бинарный являются наиболее часто используемыми форматами. Нажмите **Return**, чтобы выйти из диалогового окна.
* Нажмите **F3**, чтобы скомпилировать программу. Если вы не сохранили файл редактирования, вам будет предложено сделать это сейчас. Сохраните его как SAMPLE.C. Откроется диалоговое окно, запрашивающее адреса памяти. Введите в поле адреса адрес ПЗУ, по которому вы хотите запустить код (он почти всегда будет равен нулю), и введите в поле адреса ОЗУ начало своей ОЗУ. Если вы используете Z180, вам также нужно будет ввести физический адрес ОЗУ. Он должен быть линейным адресом в адресном пространстве Z180, в котором располагается ОЗУ. В этом случае первый адрес ОЗУ может иметь любое значение ниже 64 Кбайт, но должен быть выбран таким, чтобы избежать нижней памяти, где располагается ПЗУ.

Нажмите **Enter**, чтобы выйти из диалогового окна. HPDZ скомпилирует программу. Любые найденные ошибки остановят компиляцию, и ошибки будут перечислены в окне, которое появится внизу экрана. Курсор в окне редактирования будет расположен на строке с ошибкой. Исправьте ошибку, затем нажмите **F3** снова. Вам не придется повторно вводить адреса памяти.

По завершении компиляции выходной файл с именем SAMPLE.HEX (или SAMPLE.BIN для двоичного) будет находится в текущем каталоге.

* Выйдите из HPD, нажав ALT-Q.

## Использование ZC

Чтобы использовать ZC для компиляции примера программы, сначала нужно создать файл содержащий программу. Можно использовать любой известный текстовый редактор, если он может создать обычный файл ASCII. Команда DOS EDIT удовлетворительная. Вызовите файл SAMPLE.C. Чтобы выполнить ZC, введите:

**ZC SAMPLE.C**

Если вы компилируете для Z180, то должны добавить опцию -180, например.

**ZC -180 SAMPLE.C**

Она генерирует код для системы команд Z180 и запрашивает физический адрес ОЗУ, а также другие необходимые адреса.

Если вы правильно ввели пример программы, сообщения об ошибках не должны появляться. Если вы получаете сообщения об ошибках, отредактируйте программу, чтобы исправить ошибки, и перекомпилируйте с помощью ZC, как раньше. Затем ZC запросит некоторые адреса памяти, как показано в следующем примере:

C:\>**zc -180 sample.c**

HI-TECH C COMPILER (Z80/Z180/64180) V7.60

Copyright (C) 1984-1996 HI-TECH Software

Serial no: CZ80-12345; Licensed to:

Alfred

Invalid or no memory addresses specified:

Use a -Arom,ram,ramsize,ramphys,nvram option, e.g.

-A8000,6000,2000,10000,4000

specifies ROM at 8000(hex) and 8K bytes of RAM at physical 10000(hex) mapped into common area 1 at 6000(hex)

ROM (vectors) address: **0**

RAM address: **8000**

RAM size(default 800 hex):

Physical address of RAM: **10000**

Non-volatile RAM address (if used):

Next time use the option -A0,8000,800,10000

Linking:

Objtohex: .........................

Memory Usage Map:

User: 0069H - 0102H 009AH (154) bytes

CODE: 0000H - 0068H 0069H (105) bytes

CODE: 0103H - 015AH 0058H (88) bytes

RAM: 8000H - 8023H 0024H (36) bytes

Необходимо отметить некоторые моменты:

* Адрес ПЗУ указан равным **0**. Вряд ли вы укажете здесь какой-либо другой, если только не собираетесь запускать код в ОЗУ на плате с монитором в ПЗУ.
* Адрес ОЗУ задан равным **8000**. Это шестнадцатеричный адрес ОЗУ который отображается при помощи декодирования аппаратного адреса, или тот который вы хотите отобразить при помощи MMU в системе Z180.
* Размер оперативной памяти был оставлен по умолчанию **800** (шестнадцатеричный). Это достаточно для этого примера программы, но в целом вы должны указать здесь реальный размер ОЗУ, который доступен для скомпилированной программы.
* Была использована опция -180, поэтому, драйвер запросил физический адрес ОЗУ. Он необходим для Z180, чтобы разрешить код для настройки MMU во время выполнения. В этом примере физический адрес ОЗУ был определен равным **10000**. Вы должны ввести здесь фактический адрес ОЗУ. Если вы используете Z180, где ОЗУ и ПЗУ отображены в первые 64 Kбайта, то обычно лучше задать значения фактического адрес ОЗУ для адреса ОЗУ и физические адреса ОЗУ.
* Энергонезависимая память (Non-volatile RAM, сокращенно NVRAM) не была определена. Если вы используете NVRAM, введите здесь физический адрес NVRAM. Переменные, определенные с помощью ключевого слова persistent, хранятся в памяти NVRAM.
* ZC напечатал краткий отчет об использовании памяти. Убедитесь, что адреса в нем соответствуют указанным адресам памяти.

### Выбор формата выходного файла

Компилятор поддерживает различные форматы выходного файла. Обычно при программировании ППЗУ для ввода используются два: Intel HEX и двоичный файл. По умолчанию генерируется Intel HEX, но он может быть изменен в меню **ROM** **Output** **File...** (Выходной файл ПЗУ...) выбором пункта меню в меню опций HPDZ, или с помощью одной из следующих опций ZC:

|  |  |
| --- | --- |
| **-MOTOROLA** | Производит шестнадцатеричный файл Motorola S1/S9; |
| **-BIN** | Производит бинарный выходной файл; |
| **-UBROF** | Производят IAR Ubrof файл; |
| **-AAHEX** | Производят шестнадцатеричные записи с символами для эмуляторов American Automation; |
| **-TEKHEX** | Производят выходной шестнадцатеричный файл Tektronix. |

## Выполнение вашей программы

После компиляции программы у вас будет файл SAMPLE.HEX в текущем каталоге. Каким образом вы получите его на своем оборудовании зависит от того, с чем вам придется работать, но, вообще говоря, вам понадобится программатор ППЗУ или внутрисхемный эмулятор, который позволит вам загрузить программу в память вашей целевой системы. Точные процедуры для этого выходят за рамки настоящего руководства.

# Учебные руководства

Ниже приведены учебные руководства, которые помогут вам понять и использовать кросс-компиляторы HI-TECH C. Эти учебники следует читать вместе с соответствующими разделами руководства, поскольку они дают общий обзор некоторых аспектов компилятора. Некоторые из приведенных здесь руководств являются общими для всех компиляторов HI-TECH C и могут содержать информацию, не относящуюся к используемому компилятору.

## Обзор процесса компиляции

В этом учебном пособии дается обзор процесса компиляции, выполняемого компилятором HI-TECH C, с точки зрения обработки входных исходных файлов. Обсуждается происхождение файлов, создаваемых компилятором, а также их содержимое и назначение.

### Компиляция

При компиляции программы, задействуются многие независимые приложения, действиями которых управляют или *драйвер командной строки* (CLD) или *драйвер HPD*[[2]](#footnote-2) (HPD). В любом случае, драйвер HPD или CLD выбирает опции, определенные программистом (пункты меню в случае HPD или параметры командной строки для CLD), чтобы определить, какое из внутренних приложений должно быть выполнено и какие опции должны быть отправлены каждому из них. Под термином *компилятор*, подразумевается весь набор приложений и драйверов, задействованных в процессе. Точно так же *компиляция* относится к полному преобразованию от ввода до вывода компилятора. В этом документе каждое приложение и его функция рассматриваются более подробно.

Драйверы компилятора используют несколько файлов для хранения опций и информации, используемой в процессе компиляции. Типы этих файлов приведены в [Таблице 2-1](#t21) на странице 9. Драйвер HPD хранит опции компилятора в файле проекта, который имеет расширение ".prj". Сам HPD хранит свои собственные конфигурационные настройки в файле ".ini", например, **HPD51.ini** в каталоге **bin**. Этот файл хранит информацию, например, такую как значения цветов и параметры настройки мыши. Пользователи, использующие CLD, могут сохранить параметры командной строки в пакетных файлах DOS.

Таблица 2‑1 Конфигурационные файлы

|  |  |  |
| --- | --- | --- |
| **Расширение** | **Имя** | **Содержимое** |
| **.prj** | Файл проекта | Опции компилятора сохраненные драйвером HPD |
| **.ini** | Файл инициализации HPD | Настройки среды HPD |
| **.bat** | Командный файл | Опции драйвера командной строки, сохраненные как пакетный файл DOS |
| **.ini** | Информационный файл микросхемы | Информация о семействе микросхем |

Некоторые компиляторы поставляются с информационными файлами микросхем, которые описывают распределение памяти различных типов микросхем. При необходимости этот файл может быть отредактирован, чтобы создать новые типы микросхем, которые затем могут быть выбраны с помощью соответствующего параметра командной строки из меню **Select processor...** (Выбор процессора...). Этот файл будет также иметь расширение ".ini" и обычно находится в каталоге **lib**.

В следующих разделах процесс компиляции рассматривается с точки зрения того, что происходит на каждом этапе и файлах, которые в них участвуют. [Рисунок 2-1](#r21) на странице 11 содержит блок-схему внутренних этапов компилятора HI-TECH и таблицы типов файлов используемые в тексте этого руководства, которые перечисляют расширение файлов[[3]](#footnote-3), используемые файлами различных форматов и информацию, которую содержит файл.

Обратите внимание, что некоторые более старые компиляторы HI-TECH включают в себя не все описанные ниже приложения.

Внутренние приложения, генерируют выходные файлы и передают их в следующее приложение, как показано на рисунке. Стрелки от одного приложения (нарисованные как эллипсы) к другому осуществляется через временные файлы, имеющие неописательные имена, такие как $$003361.001. Эти файлы временно хранятся в каталоге, на который указывает переменная TEMP окружения DOS. Такая переменная создается командой **SET** DOS. Эти файлы автоматически удаляются драйвером после завершения компиляции.

### Ввод компилятора

Для создания программы пользователь предоставляет компилятору несколько вещей: входные файлы и опции компилятора, как при использовании CLD, так и HPD. Компилятор принимает входные файлы нескольких различных типов. Они описываются ниже.

Вполне возможно, и даже в большом количестве проектов, единственными файлами, предоставленными пользователем, являются *исходные файлы* C и, возможно, сопутствующие файлы заголовков. Предполагается, что любой, кто использует наш компилятор, знаком с синтаксисом языка Си. В противном случае имеется бесконечный выбор литературы, охватывающий эту тему. Исходные файлы C, обрабатываемые компилятором HI-TECH, должны использовать расширение ".c", так как это расширение используется драйвером для определения типа файла. Исходные файлы C могут быть перечислены в любом порядке в командной строке при использовании CLD или введены в диалоговом окне **Source file list**... (Список исходных файлов...) при использовании HPD.

*Заголовочный файл* обычно представляет собой файл, который содержит информацию, связанную с программой, но который непосредственно не производит исполняемый код при компиляции. Обычно они включают объявления (в противоположность определениям) для функций и типов данных. Эти файлы включаются в исходный код C директивой препроцессора include и часто называются *включаемыми файлами*. Так как ссылка команды на заголовочные файлы содержит имя и расширение файла (и возможно путь), нет никаких ограничений относительно того, каким может быть это имя, несмотря на то, что соглашение диктует расширение ".h".

Хотя исполняемый код C может быть включен в исходный файл, предполагается, что файл с расширением ".h" содержит неисполняемый контент. Любые исходные файлы C, которые должны быть включены в другие исходные файлы, должны по-прежнему иметь расширение ".c". В любом случае лучше избегать практики включения одного исходного файла в другой, так как это затрудняет структурирование кода, и теряется много преимуществ компилятора, способного обрабатывать несколько файлов с исходными кодами. Заголовочные файлы также могут быть включены в файлы ассемблера. Опять же, рекомендуется, чтобы файлы содержали только объявления ассемблера.

Рисунок 2‑1 Обзор компиляции



Таблица 2‑2 Типы входных файлов

|  |  |  |
| --- | --- | --- |
| **Расширение** | **Имя** | **Содержимое** |
| **.c** | Исходный файл C | C источник, соответствующий стандарту ANSI, возможно, с расширениями, допускаемыми HI-TECH C |
| **.h** | Заголовочный файл | Объявления C или ассемблера |
| **.as** | Ассемблерный файл | Ассемблерный источник, соответствующий формату ассемблера HI-TECH |
| **.obj** | Объектный файл | Предварительно скомпилированный C или ассемблерный источник в перемещаемый объектный файл HI-TECH |
| **.lib** | Библиотечный файл | Предварительно скомпилированный C или ассемблерный источник в формате библиотеки HI-TECH |

Компиляторы HI-TECH поставляются со многими заголовочными файлами, которые хранятся в отдельном каталоге дистрибутива. Обычно заголовочные файлы, написанные пользователем, помещаются в каталог, содержащий исходные тексты программы. В качестве альтернативы они могут быть помещены в каталог, который можно найти с помощью опции **-I** (**CPP include paths...**).

*Ассемблерный файл* содержит ассемблерные *мнемоники*, специфические для процессора для которого компилируется программа. Ассемблерные файлы могут быть получены из ранее скомпилированных исходных файлов C в ассемблер или написаны собственноручно и являться произведением искусства, созданным программистом. В любом случае эти файлы должны соответствовать формату, ожидаемому ассемблером HI-TECH, являющегося частью компилятора. Зависимость от процессора делает файлы ассемблера полностью непереносимыми, и их нужно избегать, если может быть написан источник C для выполнения поставленной задачи. Ассемблерные файлы должны иметь расширение ".as", поскольку оно используется драйвером компилятора для определения типа файла. Ассемблерные файлы могут быть перечислены в любом порядке в командной строке при использовании CLD или введены в диалоговое окно **Source file list...** (Список исходных файлов...) при использовании HPD, вместе с исходными файлами C.

В качестве входных данных драйверам компилятора могут также быть переданы предварительно скомпилированные *объектные файлы* HI-TECH. Эти файлы будут рассмотрены ниже в [Разделе 2.1.2.1](#_Шаги_перед_компоновкой) на странице 13. Эти файлы должны иметь расширение ".obj". Объектные файлы могут быть перечислены в командной строке в любом порядке при использовании CLD или введены в диалоговом окне **Object file list...** (Список объектных файлов...) при использовании HPD. Вы не должны вводить здесь имена объектных файлов, которые были уже скомпилированы из исходных файлов в проекте, включайте только объектные файлы, которые были предварительно скомпилированы и не имеют соответствующего источника в проекте. Например, должны быть перечислены такие, как файл времени выполнения.

Подпрограммы, используемые программой, обычно могут быть скомпилированы в файл, который называется *библиотечным файлом*. Эти файлы более удобны в обращении и легко доступны компилятору. Компилятор может принимать файлы библиотеки напрямую как и другие исходные файлы. Расширение ".lib" указывает тип файла и таким образом, файлы библиотек нужно называть таким образом. Файлы библиотеки могут быть перечислены в любом порядке в командной строке при использовании CLD или введены в диалоговое окно **Library file list...** (список файлов библиотек...) при использовании HPD.

Библиотечные функции HI-TECH поставляются предварительно скомпилированными в формате библиотек и хранятся в специальном каталоге в вашем дистрибутиве.

#### Шаги перед компоновкой

Из всех типов файлов, которые могут быть приняты компилятором, именно исходные файлы C требуют наибольшей обработки. В первую очередь рассмотрим этапы компиляции исходных файлов C.

Для каждого исходного файла C приложением **clist**создается *файл листинга* C. Файлы листинга содержат исходные строки C до выполнения любой обработки с предшествующим номером строки. [Таблица 2-3](#t23) на странице 13 показывает листинг C небольшой тестовой программы названной **main.c**.

Таблица 2‑3 Вывод листинга

|  |  |
| --- | --- |
| **Исходный код С** | **Листинг С** |
| #define VAL 2  int a, b = 1;  void  main(void)  {  /\* set starting value \*/  a = b + VAL;  } | 1: #define VAL 2  2:  3: int a, b = 1;  4:  5: void  6: main(void)  7: {  8: /\* set starting value \*/  9: a = b + 2;  10: } |

Исходные файлы C также передаются *препроцессору* **cpp**. Это приложение готовит источник C для последующей интерпретации. Задачи, выполняемые **cpp**, включают удаление из исходного кода комментариев и множественных пробелов (например, символов табуляции, используемых в отступах), и выполнение любых предпроцессорных директив в исходном коде. Директивы могут, например, заменять макросы заменяющим их текстом или удалять текст источника, если определенные условия не верны. Препроцессор также копирует заголовочные файлы, независимо от того, предоставлены они пользователем или компилятором в исходный код. В [Таблице 2-4](#t24) на странице 14 показан вывод препроцессора тестовой программы.

Вывод препроцессора является источником C, но он может содержать код, который был включен препроцессором из других файлов и содержит только код, для которого препроцессор оценивает конкретные условия как истина. Вывод препроцессора часто упоминается как *модуль* или *модуль трансляции*. Термин "модуль" иногда используется, чтобы описать фактический исходный файл, из которого создается "истинный" модуль. Это не совсем корректно, но смысл достаточно ясен.

Последующая генерация кода использует выходной модуль **cpp**, а не источник C и поэтому, должны быть предприняты специальные шаги для согласования ошибок и их позиции в необработанных исходных файлах C. Строка '**# 1 main.c**' в выводе препроцессора включена препроцессором для указания имени файла и номера строки в исходном файле C, которая соответствует этой позиции. Обратите внимание, что в этом примере комментарий и макроопределение были удалены, но для сохранения информации о нумерации строк их место занимают пустые строки.

Таблица 2‑4 Вывод препроцессора

|  |  |
| --- | --- |
| **Исходный код С** | **Вывод препроцессора** |
| 1: #define VAL 2  2: int a, b = 1;  3: void main(void)  4: {  5: /\* set starting value \*/  6: a = b + VAL;  7: } | 0: # 1 "main.c"  1:  2: int a, b = 1;  3: void main(void)  4: {  5:  6: a = b + 2;  7: } |

Как и остальными приложениями компилятора, препроцессором управляет драйвер компилятора (CLD или HPD). Типичная информация, которую драйвер предоставляет препроцессору, включает каталоги, в которых следует искать заголовочные файлы, включаемые в исходный файл и размер основных объектов C (таких как **int**, **double**, **char** \* и т.д.) используя опции **-S**, **-SP**, чтобы препроцессор мог оценить директивы препроцессора, которые содержат выражение **sizeof(type)**. Выходные данные препроцессора обычно не видны, если пользователь не использует опцию **-pre**, в этом случае выходные данные компилятора могут затем быть перенаправлены в файл.

Таблица 2‑5 Промежуточные и вспомогательные файлы

|  |  |  |
| --- | --- | --- |
| **Расширение** | **Имя** | **Содержимое** |
| **.pre** | Предварительно обработанный файл | Источник на языке C или ассемблере после этапа предварительной обработки |
| **.lst** | Файл листинга C | Источник на языке C с номерами строк |
| **.lst** | Листинг ассемблера | Источник C с соответствующими ассемблерными инструкциями |
| **.map** | Файл распределения памяти | Информация о символах и перемещении psect сгенерированные компоновщиком |
| **.err** | Файл ошибок | Предупреждения и ошибки компилятора, возникшие при компиляции |
| **.rlf** | Листинг модификации адресов | Информация, необходимая для обновления файл списка с абсолютными адресами |
| **.sdb** | Символьный файл отладки | Имена объектов и типы для модуля |
| **.sym** | Файл символов | Абсолютные адреса символов программы |

Вывод **cpp** передается синтаксическому анализатору **p1**. *Синтаксический анализатор* запускает первую из трудоемких операций, связанную с превращением описания программы, написанной на языке C, в исполняемый файл, состоящий из инструкций ассемблера. Синтаксический анализатор сканирует исходный код C, проверяя его корректность, и затем заменяет выражения C их измененной формой. (Последующее описание генерации кода не требуется, для понимания как использовать компилятор HI-TECH, но было включено для любознательных читателей.)

Например, выражение '**a = b + 2**' перестраивается в префиксную нотацию '**= + b 2**'. Эта нотация может легко быть интерпретирована как дерево с '**=**' в вершине, '**a**' и '**+**' ветви ниже ее, и '**b**' и '**2**' являются подветвями сложения. Вывод синтаксического анализатора для нашей маленькой программы C показан в [Таблице 2-6](#t26) на странице 15. В источнике C выделен оператор присваивания, а также результат, который генерирует синтаксический анализатор для этого оператора. Обратите внимание, что глобальные символы в выводе синтаксического анализа уже имеют символ подчеркивания, который был предварительно присвоен их имени. С этого момента ссылка на них будет создаваться с использованием этих символов. Другие символы в этой выделенной строке относятся к константе. Стандарт ANSI утверждает, что константа 2 в источнике должна интерпретироваться как целое со знаком. Синтаксический анализатор гарантирует это, путем приведения значения константы. Символ **->** представляет приведение и, 'i представляет тип. В этом выводе можно увидеть нумерацию строк, объявления переменных и начало и конец определения функции.

Таблица 2‑6 Вывод анализатора

|  |  |
| --- | --- |
| **Исходный код С** | **Вывод анализатора** |
| 1: #define VAL 2  2: int a, b = 1;  3: void main(void)  4: {  5: /\* set starting value \*/  6: **a = b + VAL;**  7: } | Version 3.2 HI-TECH Softwa...  "2 P:H0.C  [v \_a `i 1 e ]  [v \_b `i 1 e ]  [i \_b  -> 1 `i  ]  "4  [v \_main `(v 1 e ]  {  [e :U \_main ]  "6  **[e = \_a + \_b -> 2 `i ]**  "7  [e :U 1 ]  } |

Именно синтаксический анализатор отвечает за обнаружение значительной части ошибок в исходном коде. Эти ошибки будут связаны с синтаксисом исходного кода. Синтаксический анализатор также выдает предупреждения, если код необычен.

Синтаксический анализатор передает свой вывод непосредственно к следующему этапу в процессе компиляции. Драйвер не имеет опций, которые заставляли бы анализатор генерировать выходные файлы с синтаксическим анализом, поскольку эти файлы не содержат полезной информации для программиста.

Теперь самая сложная часть компиляции: генерация кода. *Генератор кода* преобразует вывод синтаксического анализатора в ассемблерные мнемоники. Это первый шаг процесса компиляции, который зависит от процессора. В то время как все препроцессоры и синтаксические анализаторы HI-TECH имеют одно и то же имя и фактически представляют собой одно и то же приложение, генераторы кода имеют конкретное, основанное на имени процессора, например **cgpic** или **cg51**.

Генератор кода использует ряд правил или производств для создания ассемблерного вывода. Чтобы понять, как производство работает, рассмотрим следующую аналогию производства раньше генерировавшую код для выражения сложения в нашей тестовой программе. "Если Вы можете получить один операнд в регистр", и "один операнд - целая константа" тогда, вот код, который выполнит 2-байтовое сложение их. Здесь, каждая заключенная в кавычки строка представляет собой вспомогательное производство, которое должно быть соответствующим. Первая строка будет попытаться получить содержимое **\_a** в регистр, путем дальнейшего сопоставления вспомогательных производств. Если это не возможно, это производство не может использоваться, и будет рассматриваться другое. Если все вспомогательные производства могут быть выполнены, то код, который они производят можно свести воедино в порядке, указанном в дереве производств. На самом деле не все производства генерируют код, но необходимы для процесса сопоставления.

Если соответствующие производства или вспомогательные производства не найдены, генератор кода выдаст ошибку "Can't generate code for this expression" (Генерация кода для этого выражения невозможна). Это означает, что исходный код C является законным и генератор кода попытался произвести код ассемблера для него, но в данном контексте, нет производств, которые могут соответствовать выражению.

Таблица 2‑7 Вывод генератора кода

|  |  |
| --- | --- |
| **Исходный код С** | **Код ассемблера (Z80)** |
| 1: #define VAL 2  2: int a, b = 1;  3: void main(void)  4: {  5: /\* set starting value \*/  6: **a = b + VAL;**  7: } | psect text  global \_main  \_main:  ld hl,(\_b)  inc hl  inc hl  ld (\_a),hl |

Обычно полная реализация генератора кода может иметь около 800 производств. Было около дюжины совпадающих производств для генерации кода для оператора, выделенного в [Таблице 2-7](#t27) на странице 16 для генератора кода Z80. Он проверил около 70 возможных производств, прежде чем найти решение. Точный процесс генерации кода слишком сложен, для описания в этом документе и не требуется для эффективного использования компилятора.

Пользователь может остановить процесс компиляции после генерации кода путем задания драйверу опции **-s** (компиляция в .as). В этом случае генератор кода оставит файлы ассемблера с расширением ".as".

[Таблица 2-7](#t27) на странице 16 показывает вывод, сгенерированный генератором кода Z80. Показан ассемблерный код только для открывающей фигурной скобки **\_main** и выделенной исходной строки. Этот вывод будет отличаться для других компиляторов и опций компилятора.

Генератор кода может также создать отладочную информацию в форме файла ".sdb". Эта операция включается при помощи опции -g (**Source** **level** **debug** **info** (Информация отладки исходного уровня)). Для каждого компилируемого модуля создается один файл отладки. Эти файлы ASCII содержат информацию о символах, определенных в каждом модуле, и могут использоваться программами отладки. [Таблица 2-6](#t26) на странице 15 показывает файлы отладки, которые могут быть созданы компилятором на различных этапах компиляции. Некоторые из выходных форматов также содержат отладочную информацию в дополнение к коду и данным.

Генератор кода дополнительно выполняет еще одну задачу: оптимизацию. Компиляторы HI-TECH поставляются с оптимизаторами с несколькими различными уровнями. Генератор кода отвечает за *глобальную оптимизацию*, которая может быть включена, используя опцию **-Zg** (Глобальная оптимизация). Эта оптимизация выполняется на проанализированном источнике. Среди прочего, этот этап оптимизации выполняется каждый раз, когда возможно, выделяет регистры переменным и ищет константы, которые используются последовательно в исходном коде, для избегания излишних перезагрузок этих значений без необходимости.

Файлы ассемблера являются первыми файлами в процессе компиляции, которые ссылаются на psect или разделы программы. Генератор кода будет генерировать директивы psect, в которых будут располагаться код и данные.

Затем вывод генератора кода передается *ассемблеру*, который преобразует мнемоническое ASCII представление инструкций процессора в двоичный *машинный код*. Ассемблер каждого компилятора имеет имя зависящее от процессора, например, **aspic** или **asxa**. Ассемблерный код также содержит ассемблерные директивы, которые будут выполняться ассемблером. Некоторые из этих директив определяют константы ПЗУ, некоторые определяют psect, и некоторые объявляют глобальные символы.

Ассемблеру необязательно предшествует оптимизация сгенерированного ассемблера. Это - *локальная оптимизация*. В некоторых компиляторах HI-TECH локальный оптимизатор содержится в самом ассемблере, например, ассемблере PIC, однако другие содержат отдельное приложение оптимизации, которое запускается, до выполнения ассемблера, например, **opt51**. Локальная оптимизация выполняется отдельно в ассемблерном коде, полученном из каждой отдельной функции.

В дополнение к локальному оптимизатору сам ассемблер может включать отдельный шаг ассемблерной оптимизации, который по возможности пытается заменить длинные ветви короткими ветвями. Опция **-O** включает оба ассемблерных оптимизатора, даже если они выполнены в виде отдельных приложений, однако HPD включает пункты меню для обоих этапов оптимизации: **Peephole optimization** (Локальная оптимизация) и **Assembler** **optimization** (оптимизация ассемблера). Если локальный оптимизатор является частью ассемблера, пункт оптимизации ассемблера в HPD не оказывает никакого эффекта.

Выводом ассемблера является объектный файл. *Объектный файл* представляет собой отформатированный двоичный файл, который содержит машинный код, данные и другую информацию, относящуюся к модулю, из которого он был сгенерирован. Объектные файлы бывают двух основных типов: *перемещаемые* и *абсолютные* объектные файлы. Несмотря на то, что оба содержат машинный код в двоичной форме, в перемещаемых объектных файлах адреса не могут иметь абсолютных значений. Двоичный машинный код сохраняется в виде блока для каждого psect. Любые адреса в этой области временно сохранены как 00h. Отдельная информация о перемещении в объектном файле указывает, где эти неразрешенные адреса располагаются в psect и что они представляют. Объектные файлы также содержат информацию относительно любых определенных psect, чтобы компоновщик мог расположить их правильно.

Таблица 2‑8 Вывод ассемблера

|  |  |
| --- | --- |
| **Исходный код С** | **Перемещаемый объектный файл** |
| 1: #define VAL 2  2: int a, b = 1;  3: void main(void)  4: {  5: /\* set starting value \*/  6: **a = b + VAL;**  7: } | 11 TEXT 22  text 0 13  99 08 00 00 88 10 A9 12 8E 00 00 D6 80  12 RELOC 63  2 RPSECT data 2  9 COMPLEX 0  Key: direct  0x7>=(high bss)  9 COMPLEX 1  ((high bss)&0x7)+0x8  10 COMPLEX 1  low bss |

Объектные файлы, произведенные ассемблером, следуют формату, который является стандартным для всех компиляторов HI-TECH, но очевидно, что их содержимое зависит от машины. [Таблица 2-8](#t28) на странице 18 показывает несколько разделов формата HI-TECH перемещаемого объектного файл, который был преобразован в ASCII представление, используя исполняемую программу DUMP, которая поставляется с компилятором. Выделенная исходная строка представлена выделенным машинным кодом в объектном файле. Этот код помещается в psect названный **text**. Подчеркнутые байты в объектном файле являются адресами, которые пока неизвестны и заменены нулями. Строки после **text** psect в объектном файле отображают информацию, используемую для разрешения адресов, необходимую компоновщику. Здесь показаны два байта, начинающиеся со смещения 2, и два одиночных байта со смещением 9 и 10, и, как можно видеть, их адрес будет содержаться в адресе, полученном из позиции psect **data** и **bss**, соответственно.

Если определена опция **-asmlist** (**Generate** **assembler** **listings** (генерация листингов ассемблера)), ассемблер генерирует файл листинга ассемблера, который содержит как оригинальные строки исходного кода на языке C, так и код ассемблера, созданный для каждой строки. Вывод листинга ассемблера показан в [Таблице 2-9](#t29) на странице 19. Неразрешенные адреса указаны как нулевые с маркерами неразрешенных адресов "'" и "\*" которые используются для обозначения того, что значения не являются абсолютными. Обратите внимание, что код помещается начиная с нулевого адреса в новый psect **text**. Весь psect будет перемещен компоновщиком.

Некоторые ассемблеры HI-TECH также создают *файл с перемещаемым листингом* (расширение: ".rlf")[[4]](#footnote-4). Он содержит адресную информацию, которая может быть прочитана компоновщиком и использована для обновления файла листинга ассемблера, если такой файл был создан. После связывания в файле листинга ассемблера будут удалены маркеры неразрешенных адресов и адреса заменены абсолютными адресами.

Вышеупомянутая последовательность действий: предварительная обработка, синтаксический анализ, генерация кода и ассемблирование, выполняются для каждого исходного файла C, переданного драйверу поочередно. Ошибки в коде отражаются по мере их обнаружения. Если файл не может быть обработан компилятором из-за ошибки, драйвер прекращает компиляцию модуля после завершения приложения, вызвавшего ошибку и продолжает обработку следующего переданному ему файла, снова запустив приложение **clist**.

Таблица 2‑9 Листинг ассемблера

|  |  |
| --- | --- |
| **Исходный код С** | **Листинг ассемблера** |
| #define VAL 2  int a, b = 1;  void  main(void)  {  /\* set starting value \*/  **a = b + VAL;**  } | 10 0000' psect text  11 0000' \_main:  12 0000' psect text  13 0000' C3 000C' jp f30  14 0003' f31:  15 ;main.c: 9: a = b + 2;  16  17 global \_a  18 global \_b  19 0003' 2A 0000' ld hl,(\_b)  20 0006' 23 inc hl  21 0007' 23 inc hl  22 0008' 22 0000' ld (\_a),hl  23 ;main.c: 10: }  24  25 000B' l1:  26 000B' C9 ret |

Любые файлы ассемблера, передаваемые драйверу, не требуют такой обработки, как исходные файлы на языке C, но они должны быть ассемблированы. Драйвер компилятора передает любые файлы ".as" непосредственно ассемблеру. Если пользователь указывает опцию **-p** (**Pre-process assembler files** (предварительная обработка ассемблерных файлов)), файлы ассемблера сначала передаются препроцессору C, что позволяет использовать все предпроцессорные директивы в коде ассемблера. Затем вывод препроцессора передается ассемблеру.

Объектные файлы и библиотеки, переданные компилятору, уже скомпилированы и не обрабатываются на первых этапах компиляции. Они не используются до этапа связывания, который объясняется ниже.

Если вы используете HPD, *информацию о зависимостях* можно сохранить относительно каждого файла источника и заголовка, щелкнув по переключателю **Save dependency information** (Сохранить информацию о зависимостях). Когда этот параметр включен, драйвер HPD определяет, какие файлы в проекте необходимо перекомпилировать с дат изменения входных исходных файлов. Если исходный файл не был изменен, используется существующий объектный файл.

#### Этап компоновки

Формат объектных файлов снова не зависит от процессора, поэтому компоновщик и другие приложения, обсуждаемые ниже, являются общими для всех компиляторов HI-TECH. Имя компоновщика - **hlink**.[[5]](#footnote-5)

Компоновщик решает много задач. Компоновщик отвечает за объединение всех объектных и библиотечных файлов в один файл. Файлы, обрабатываемые компоновщиком, включают все объектные файлы, скомпилированные из входных исходных C и ассемблерных файлов, плюс любые объектные файлы, или файлы библиотек переданные драйверу компилятора, плюс любые объектные файлы периода выполнения и файлы библиотек, предоставленные драйвером. Компоновщик также выполняет *группировку* и *перемещение* psect, содержащихся во всех переданных ему файлах, используя относительно сложный набор опций компоновщика. Компоновщик также разрешает именам символов быть абсолютными адресами после того, как перемещение сделало возможным определить, где объекты должны храниться в ПЗУ или ОЗУ. Затем компоновщик корректирует ссылки на эти символы - процесс, известный как *привязка адресов*. Если адрес символа оказывается слишком большим, чтобы вписаться в пространство в инструкции, сгенерированной генератором кода, возникает ошибка "fixup overflow" (сегментное переполнения).

Компоновщик также может генерировать файл распределения памяти (map file), который содержит подробные сведения о расположении сегментов psect и адресов, присвоенных символам. Компоновщик также может создать файл символов. Эти файлы имеют расширение ".sym" и генерируются при использовании опции **-g** (**Source** **level** **debug** **info** (Информация отладки исходного уровня)). Этот файл символов основан на ASCII и содержит информацию для всей программы. Адреса являются абсолютными, поскольку этот файл генерируется после этапа связывания.

Вывод компоновщиков для компиляторов, которые компилируют на компьютерах с операционной системой, является истинными исполняемыми объектными файлами, которые можно загрузить и запустить. Однако процесс компиляции не полностью завершен для кросс-компилятора. Хотя объектный файл, созданный **hlink**, содержит всю информацию, необходимую для запуска программы, программа должна быть каким-то образом перенесена с хост-компьютера на встроенное аппаратное обеспечение.

Существует много стандартных форматов, которые были созданы для такой задачи. Эмуляторы и программаторы микросхемы часто могут принимать некоторые из этих форматов. Форматы Motorola HEX (S запись) или Intel HEX являются стандартными форматами. Это форматы ASCII, позволяющие простой просмотр любым текстовым редактором. Они включают информацию о *контрольной сумме*, которая может использоваться программой, загружающей файл, чтобы гарантировать, что он был передан без ошибок. Эти форматы содержат адресную информацию, которая позволяет исключить из файла области, которые не содержат данных. Это позволяет сделать эти файлы существенно меньше, чем, например, двоичный файл.

Приложение **objtohex** отвечает за создание выходного файла, запрошенного пользователем. Он принимает абсолютный объектный файл, произведенный компоновщиком, и производит вывод под управлением драйвера компилятора. Приложение **objtohex** может произвести несколько различных форматов, чтобы удовлетворить большинство систем разработки. Выходные типы, доступные в большинстве компиляторов HI-TECH, показаны в [Таблице 2-10](#t210) на странице 21.

При некоторых обстоятельствах требуется больше чем один выходной файл. В этом случае выполняется приложение, названное **cromwell**, преобразователь форматов, для получения дополнительных выходных файлов. Например, он часто используется с компилятором PIC, для чтения файлов HEX и SYM и создания файла COD.

Таблица 2‑10 Выходные форматы

|  |  |  |
| --- | --- | --- |
| **Расширение** | **Имя** | **Содержимое** |
| **.hex** | Motorola hex | Код ASCII, формат Motorola S19 |
| **.hex** | Intel hex | Код ASCII, формат Intel |
| **.hex** | Tektronix hex | Код ASCII, формат Tektronix |
| **.hex** | American Automation hex | Код и информация о символах в двоичном формате American Automation |
| **.bin** | двоичный файл | Код в двоичном формате |
| **.cod** | Файл Bytecraft COD | Код и информация о символах в двоичном формате Bytecraft |
| **.cof** | Файл COFF | Код и информация о символах в двоичном общем формате объектных файлов |
| **.ubr** | Файл UBROF | Код и информация о символах в универсальном двоичном перемещаемом объектном формате |
| **.omf** | Файл OMF-51 | Код и информация о символах в формате объектных модулей Intel для 8051 |
| **.omf** | Расширенный файл OMF- 51 | Код и информация о символах в формате объектного модуля Keil для 8051 |

## Секции psect и компоновщик

Это учебное руководство объясняет, как компилятор разбивает код и данные объектов в программе C на различные части и затем как указать компоновщику их место расположения в ПЗУ и ОЗУ на целевой системе.

### Секции psect

Во время выполнения генератор кода генерирует ассемблерный файл для каждого компилируемого исходного файла C. Содержимое этих ассемблерных файлов включает в себя различные секции: одни содержат ассемблерные инструкции, которые представляют источник C, другие содержат ассемблерные директивы, резервирующие место для переменных в ОЗУ, другие, содержат константы ПЗУ, определенные в источнике C, и некоторые, содержат данные для специальных объектов, таких как энергонезависимые переменные, векторы прерываний и слова конфигурации, используемые процессором. Так как исходных файлов может быть несколько, подобные секции ассемблера будут рассредоточены по нескольким ассемблерным файлам, которые должны быть сгруппированы вместе после завершения генерации кода.

Эти различные секции ассемблера необходимо сгруппировать специальным образом: имеет смысл объединить все инициализированные значения данных в смежные блоки, чтобы их можно было скопировать в ОЗУ перемещением одного блока, а не разбрасывать их между секциями кода. То же самое относится к неинициализированным глобальным объектам, которым должно быть выделено место, которое затем очищается до запуска программы. Некоторый код или объекты должны быть помещены в определенные участки памяти, чтобы соответствовать возможностям адресации процессора. И иногда пользователь должен быть в состоянии расположить код или данные в конкретные абсолютные адреса для удовлетворения специальных требований программного обеспечения. Поэтому, генератор кода должен включать информацию, которая указывает, как обрабатывать различные ассемблерные секции и где они должны быть расположены компоновщиком позже в процессе компиляции.

Метод, используемый компилятором HI-TECH для группировки и размещения различных частей программы, заключается в том, чтобы поместить все инструкции и директивы ассемблера в отдельные, перемещаемые секции. Эти секции программы известны как psect - сокращение от *программные секции*. Затем компоновщик передает ряд параметров, которые указывают память, доступную в целевой системе, и каким образом все psect в программе должны быть расположены в этой области памяти.

#### Директивы psect

Директивы ассемблера psect (сгенерированные генератором кода или вручную включенные в другие файлы ассемблера) определяют новую psect. Общая форма этой директивы показана ниже.

**psect имя,опция,опция...**

Она состоит из слова **psect** за которым следует имя, к которому должна быть отнесена эта psect. **Имя** может быть любым допустимым идентификатором ассемблера и не должно быть уникальным. Т.е. может быть несколько psect с тем же именем, даже в одном файле. Как будет описано ниже, psect с одинаковыми именами обычно группируются компоновщиком.

**Опции** директив описаны в разделе [руководства по ассемблеру](#_Секции_программы), но некоторые из них будут описаны в этом учебном руководстве. Опции - инструкции компоновщику, которые описывают, как psect должна быть сгруппирована и перемещена в заключительный абсолютный объектный файл.

Одинаковое имя psect, подразумевает, что их содержание подобно и они должны быть сгруппированы и соединены одинаковым способом. Это позволяет помещать в памяти объекты вместе, даже если они определены в различных файлах.

После определения psect, в последующих директивах psect с тем же именем в том же модуле опции могут быть опущены. Следующий пример показывает определение двух psect и заполнение кодом и данными.

**psect text,global**

**begin:**

**mov r0,#10**

**mov r2,r4**

**add r2,#8**

**psect data**

**input:**

**ds 8**

**psect text**

**next:**

**mov r4,r2**

**rrc r4**

В этом примере psect **text** определена включая опцию уточняющую, что это - глобальная psect. В эту psect помещены три ассемблерные инструкции. Создана еще одна psect: **data**. Эта psect резервирует 8 байт памяти для данных в ОЗУ. Последняя директива psect продолжит добавление к первой psect. В этом примере в директивы psect опции были опущены, поскольку в этом файле уже была директива psect, которая определяет опции для psect с этим именем. Вышеупомянутый пример генерирует две psect. Другие ассемблерные файлы в программе могут также создать psect, которые имеют такое же имя, как эти здесь. Они будут сгруппированы компоновщиком с вышеупомянутыми в соответствии с флагами директивы psect.

#### Типы psect

Секции psect бывают трех основных типов: те, которые будут находиться постоянно в ПЗУ[[6]](#footnote-6), те, которым будет выделено место в ОЗУ после запуска программы и те, которые будут находиться в ПЗУ, но которые будут скопированы в другое зарезервированное место в ОЗУ после запуска программы. Комбинация кода, известная как код запуска (*startup*) среды выполнения (*run-time* ), psect и опции компоновщика позволяют это осуществить.

Как правило, psect, помещенные в ПЗУ содержат инструкции и постоянные данные, которые не могут быть изменены. Другим psect выделяется место в ОЗУ, только для объектов глобальных данных, которые не должны принимать какое либо ненулевое значение при запуске программы, т.е. они неинициализированные. Иные psect, имеют образ ПЗУ и место, зарезервированное в ОЗУ, для глобальных объектов изменяемых данных, которые инициализированы, то есть содержат некоторое определенное значение при начале работы программы, но это значение может быть изменено программой во время ее выполнения.

Следующий источник C показывает определения двух объектов. Объект **input** будет помещен в psect data; значение 22 будет находиться в ПЗУ и будет скопировано в область ОЗУ, выделенное для **input** кодом времени выполнения. Объект **output** не будет вносить непосредственный вклад в образ ПЗУ. Область памяти будет зарезервирована для него в ОЗУ, и эта область будет очищена кодом времени выполнения (**output** будет присвоено значение 0).

**int input = 22; // инициализированный объект**

**int output; // неинициализированный объект**

Фрагменты файла листинга ассемблера показывают, как компилятор 8051XA обрабатывает эти два объекта. Другие компиляторы могут произвести код, имеющий другую структуру. В данный момент обсуждаются управляющие флаги psect, но обратите внимание, что для инициализированного объекта **input**, генератор кода использовал директиву **dw** (определение слова), которая поместила два байта целого значения (16 и 00) в **input**, который предназначен для ПЗУ. Для неинициализированного объекта **output** место для двух байтов зарезервировано с помощью директивы ассемблера **ds**, и **output** не содержит ни каких значений.

**1 0000' psect data,class=CODE,space=0,align=0**

**2 global \_input**

**3 align.w**

**4 0000' \_input:**

**5 0000' 16 00 dw 22**

**13 0000' psect bss,class=DATA,space=1,align=0**

**14 global \_output**

**15 align.w**

**16 0000' \_output:**

**17 0000' ds 2**

Автоматические переменные и параметры функции локальны для функции, в которой они определены и поэтому обрабатываются компилятором иначе. Может быть динамически выделено место (например в стеке), в этом случае они не сохраняются компилятором в psect.

Для обозначения местоположения psect используются два адреса: адрес ссылки и адрес загрузки. *Адрес ссылки* - адрес, с помощью которого можно получить доступ к psect (и любым объектам или меткам в секции psect) во время выполнения программы. *Адрес загрузки* - адрес, по которому psect будет находиться в выходном файле, который создает образ ПЗУ, или, альтернативно, адрес по которому к psect можно получить доступ в ПЗУ.

Секции psect, расположенные в ПЗУ, для которых адреса ссылки и загрузки совпадают, находятся в ПЗУ и никогда не копируются в новое расположение. Секции psect, которым выделяется место только в ОЗУ, имеют только адрес ссылки, к ним адрес загрузки не применим. Компилятор часто делает адрес загрузки этих psect совпадающим с адресом ссылки. Так как образ ПЗУ для этих psect не сформирован, адрес загрузки бессмыслен и может быть проигнорирован. Любой доступ к объектам, определенным в этих psect выполняется, используя адрес ссылки. Секции psect, расположенные в ПЗУ, но скопированные в ОЗУ, имеют адреса ссылки и загрузки, которые обычно отличаются. Любые ссылки на символы или метки в этих psect всегда делаются, используя адреса ссылки.

## Соединение psect

После того, как генератор кода и ассемблер[[7]](#footnote-7) закончили свои задания, объектные файлы, переданные компоновщику, можно рассматривать как смесь секций psect, которые должны быть сгруппированы и расположены в доступных ПЗУ и ОЗУ. Опции компоновщика указывают объем доступной памяти, и флаги, связанные с директивой psect, указывающие, как psect должны быть обработаны.

### Группировка psect

Есть два флага директивы psect, которые влияют на группировку или слияние секций psect. Это флаги LOCAL и GLOBAL. GLOBAL является значением по умолчанию и говорит компоновщику, что секция psect должна группироваться с другой секцией psect с тем же именем для формирования единственной секции psect. Секции psect LOCAL не группируются подобным образом, если они не находятся в одном модуле. Две локальные секции psect, имеющие одно и то же имя, но которые определены в различных модулях, обрабатываются и располагаются как отдельные секции psect.

### Расположение psect

Несколько флагов директивы psect влияют на расположение секций psect в памяти. Секции psect с одинаковым именем могут быть расположены одним из двух способов: они могут быть наложены друг на друга, или размещены так, чтобы каждая занимала отдельную область памяти.

Секции psect, которые должны быть наложены, будут использовать флаг OVLRD директивы psect. На первый взгляд может показаться нелогичным совмещать секции psect, поскольку они могут уничтожить содержание других psect, при их размещении, однако бывают случаи, когда это желательно.

Один случай, где используется наложение psect, когда компилятор должен использовать временные переменные. Когда компилятор должен передать несколько объектов данных, скажем, подпрограмме с плавающей точкой, может потребоваться хранить числа с плавающей точкой во временных переменных расположенных в ОЗУ. Нежелательно иметь зарезервированное пространство, если оно не будет использоваться, поэтому подпрограммы, использующие временные объекты, также несут ответственность за определение области и резервирование пространства, в котором они будут находиться. Однако, могут вызваться несколько подпрограмм и, следовательно, присутствовать несколько временно создаваемых областей. Чтобы обойти эту проблему, psect, которые содержат директивы для резервирования пространства для объектов, определяются как наложенные, так что если определено более одного, они просто накладываются друг на друга.

Другая ситуация, где используется наложение psect - определение векторов прерываний. Код времени выполнения обычно определяет вектор сброса, но инициализация других векторов оставляется на усмотрение программиста. Векторы прерывания помещаются в отдельную psect (часто называемую vectors). Каждый вектор размещается со смещением от начала области векторов, подходящей для целевого процессора. Смещение достигается с помощью директивы ассемблера ORG, которая перемещает счетчик адреса относительно начала текущей psect. Макросы, содержащиеся в заголовочном файле <intrpt.h>, позволяют программисту определить дополнительные вектора прерываний, также помещают определенные им векторы в psect с таким же именем, но с разными смещениями, зависящими от определяемого вектора прерывания. Все эти psect сгруппированы и наложены таким образом, что вектора правильно располагаются по одному и тому же адресу - началу psect vectors. Эта объединенная psect затем размещается компоновщиком так, чтобы она начиналась в начале области векторов.

Большинство других сгенерированных компилятором psect не наложено и таким образом, каждая из них займет свое собственное уникальное адресное пространство. Обычно эти psect расположены в памяти одна за другой, однако есть несколько флагов psect, которые могут изменить расположение psect. Некоторые из этих флагов psect описаны ниже.

Флаг RELOC используется, когда psect должна быть выравнена на границе в памяти. Эта граница – число с кратным значением, определенным флагом. Флаг ABS определяет, что psect абсолютная и она должна начинаться по адресу 0h. Помните, однако, что, если имеется несколько psect, которые используют этот флаг, то после группировки только одна может на самом деле начинаться с адреса 0h, если только psect также не определены для наложения. Таким образом ABS означает, что одна из psect с этим именем будет расположена по адресу 0h, а остальные следовать в памяти за ней при условии использования любых других флагов psect.

### Опции компоновщика расположения psect

Компоновщику об установке памяти для целевой программы сообщают опции компоновщика, которые генерируются драйвером компилятора. Пользователь сообщает драйверу компилятора об используемой памяти, используя опцию -A[[8]](#footnote-8) драйвер командной строки (CLD), или через диалоговое окно **ROM & RAM addresses...** в HPD. Дополнительно linker options (опции компоновщика) указывают, как psect должны быть расположены в доступной памяти.

Опции компоновщика сначала немного сбивают с толку, но в следующем примере показано, как можно создавать параметры при разработке программы, а затем рассматриваются некоторые из конкретных схем, используемых компиляторами HI-TECH. При компиляции с помощью CLD или HPD используется полный набор опций компоновщика по умолчанию, основанный либо на значениях опции **-A**, либо на значениях диалога **ROM & RAM addresses**. В большинстве случаев опции компоновщика не требуют изменений.

#### Размещение psect по адресу

Предположим, что процессор в целевой системе может адресовать 64 Кбайт памяти, и что ПЗУ, ОЗУ и периферийные устройства совместно используют этот же блок памяти. ПЗУ помещается в верхние 16 Кбайт памяти (C000h - FFFFh). ОЗУ помещается по адресам от 0h до FFFh.

Давайте также предположим, что три объектных файла передавались компоновщику: один объектный файл периода выполнения, другие скомпилированы из исходного кода программиста на C. Каждый объектный файл содержит сгенерированную компилятором psect **text** (psect названную **text**): psect в первом файле - 100h байтов длиной, во втором файле - 200h байтов длиной; в объектном файле периода выполнения длиной 50h. Эти psect должны быть помещены в ПЗУ, и все они имеют простое определение, вырабатываемое генератором кода:

**psect text,class=CODE**

Флаг **CLASS** обычно используют с этими типами psect и рассматривается позже в этом учебном руководстве. По умолчанию эти секции psect - GLOBAL, следовательно после сканирования всех объектных файлов переданных ему, компоновщик сгруппирует все секции psect **text** вместе так, чтобы они были смежными[[9]](#footnote-9) и сформировали одну большую секцию psect **text**. Следующая опция компоновщика **-p** может использоваться для расположения psect **text** у основания ПЗУ.

**-ptext=0C000h**

Эта опция компоновщика содержит только один адрес, поскольку код, содержащийся в psect, не копируется из ПЗУ в ОЗУ на любом этапе, а адреса ссылки и загрузки одинаковы.

Компоновщик переместит сгруппированную секцию psect **text** так, чтобы она начиналась с адреса C000h. Затем компоновщик определит два глобальных символа с помощью имен: **\_\_Ltext** и **\_\_Htext** и приравняет их со значениями: C000h и C350h, которые являются начальным и конечным адресами, соответственно, сгруппированной psect **text**.

Предположим теперь, что файл времени выполнения и один из файлов программиста содержат векторы прерываний. Эти векторы должны быть расположены в правильном месте для этого процессора. Наш фиктивный процессор ожидает, что его векторы будут присутствовать между ячейками памяти FFC0h и FFFFh. Вектор сброса занимает два байта по адресу FFFEh и FFFFh, а остальные местоположения - для векторов периферийных прерываний. Код времени выполнения обычно определяет вектор сброса, используя следующий код.

**global start**

**psect vectors,ovlrd**

**org 3Eh**

**dw start**

Этот ассемблерный код создает новую psect, которая называется **vectors**. Эта psect использует флаг наложения (**OVLRD**), который сообщает компоновщику, что любые другие psect с этим именем должны быть наложены на нее, а не сцеплены с ней. Поскольку psect по умолчанию является глобальной, даже psect **vectors** из других файлов будут сгруппированы с ней. Директива **org** говорит ассемблеру продвинуть **3Eh** расположение в этой psect. Она не говорит ассемблеру помещать объект по адресу **3Eh**. Конечное место назначения вектора определяется перемещением psect, выполняемым компоновщиком позже в процессе компиляции. Используется директива ассемблера **dw**, чтобы фактически поместить слово в этом месте. Слово содержит адрес (глобального) символа **start**. (метки **start** или **powerup**, обычно ассоциируется с началом кода времени выполнения.)

В одном из исходных файлов пользователя использовался макрос **ROM\_VECTOR**, чтобы обеспечить одно из прерываний ввода-вывода со смещением **10h** в область векторов. Макрос расширяется до следующего встроенного кода ассемблера.

**global\_timer\_isr**

**psect vectors,ovlrd**

**org 10h**

**dw \_timer\_isr**

После завершения первых стадий компиляции, компоновщик сгруппирует все psect **vectors**, найденные во всех объектных файлах, но все они будут начинаться с того же адреса, т.е. размещены один поверх другого. Заключительная сгруппированная psect **vectors** будут содержать слово со смещением 10h и другое со смещением 3Eh. Пространство от 0h до смещения 0Fh и промежуток между этими двумя векторами компоновщик оставляет нетронутым. Опции компоновщика, необходимые для размещения этой psect следующие:

**-pvectors=0FFC0h**

Адрес, заданный с помощью этой опции, является начальным адресом области векторов. Директивы **org** служат для перемещения в пределах psect **vectors**, следовательно относительно этого начального адреса.

Оба файла пользователя содержат константы, которые должны располагаться в ПЗУ. Генератор кода создает следующую директиву psect, который определяет **psect**, в которой хранятся значения.

**psect const**

Компоновщик сгруппирует все эти psect **const** вместе, и они могут быть просто размещены как psect **text**. Единственная проблема: где? В данный момент psect **text** заканчивается по адресу C34Fh, таким образом, мы могли бы расположить psect const сразу после нее по адресу C350h, но если мы изменяем программу, мы каждый раз будем должны корректировать опции компоновщика. К счастью, мы можем задать опцию компоновщика как показано ниже.

**-ptext=0C000h,const**

Мы не указали адрес для psect **const**, поэтому, он по умолчанию принимает значение адреса непосредственно следующего после окончания предыдущей psect, указанного в параметре, то есть адреса сразу после окончания psect **text**. Опять же psect **const** находится только в ПЗУ, поэтому этот адрес определяет два адреса: ссылки и загрузки.

Теперь секции psect в ОЗУ. Объектные файлы пользователя содержат неинициализированные объекты данных. Генератор кода генерирует psect **bss[[10]](#footnote-10)**, используемые для хранения значений, неинициализированных объектов C. Область памяти, назначенная для psect **bss**, должна быть очищена до выполнения функции **main()**.

Во время компоновки все psect **bss** группируются и объединяются. Сгруппированная секция psect должна быть расположена в начале ОЗУ. Это легко сделать с помощью следующей опции.

**-pbss=0h**

Адрес **0h** является адресом ссылки psect. Адрес загрузки не имеет смысла, но по умолчанию будет указывать адрес ссылки. Код времени выполнения очистит область памяти, занятую psect **bss**. Этот код будет использовать символы **\_\_Lbss** и **\_\_Hbss** для определения начального адреса и длины области, которая должна быть очищена.

Оба исходных файла пользователя содержат инициализированные объекты, подобные приведенному ниже.

**int init = 34;**

Значение **34** должно быть загружено в объект **init**, прежде чем **main()** начинает выполнение. Еще одна из задач кода времени выполнения заключается в инициализации такого рода объектов. Это означает, что начальные значения должны быть сохранены в ПЗУ для использования кодом времени выполнения. Но объект является переменной, которой может быть присвоено значение, таким образом, он должен присутствовать в ОЗУ после запуска программы. Код времени выполнения должен скопировать инициализированные значения из ПЗУ в ОЗУ непосредственно перед началом **main()**. Компоновщик поместит все начальные значения в ПЗУ в точно том же порядке, в котором они появятся в ОЗУ так, чтобы код во время выполнения мог просто скопировать значения из ПЗУ в ОЗУ как единственный блок. Компоновщику нужно сообщить, где в ПЗУ эти значения должны находиться, поскольку он генерирует образ ПЗУ, но он должен также знать, куда в ОЗУ эти объекты будут скопированы, чтобы он смог оставить достаточно места для них и назначить (разрешить) адреса времени выполнения для символов в этой области.

Полные опции компоновщика для нашей программы, включая расположение psect **data**, могут выглядеть так:

**-ptext=0C000h,const**

**-pvectors=0FFC0h**

**-pbss=0h,data/const**

Т.е. psect **data** должна быть расположена в ОЗУ после конца psect **bss**. Адрес после наклонной черты указывает, что эта psect будет скопирована из ПЗУ и что ее позиция в ПЗУ должна быть сразу после конца psect **const**. Как и в случае другими psect, компоновщик определит символы **\_\_Ldata** и **\_\_Hdata** для этой psect, являющиеся начальным и конечным адресами соответственно, которые будут использоваться кодом времени выполнения для копирования сгруппированной psect **data**. Однако, для любых других psect, имеющих различные адреса ссылки и загрузки, определяется еще один символ, в данном случае: **\_\_Bdata**. Это - адрес загрузки psect **data** в ПЗУ.

#### Исключительные случаи

Компилятор PIC обрабатывает данные psect data несколько иначе. Он фактически определяет две отдельные psect: одна psect data для образа ПЗУ, другая для копии в ОЗУ. Это вызвано тем, что длина образа ПЗУ отличается от длины psect в ОЗУ. (ПЗУ шире, чем ОЗУ и значения сохраненные в ПЗУ, могут быть закодирован как инструкции **retlw**.) В этом случае опции компоновщика будут содержать две отдельные записи для обоих psect вместо одной psect с указанием различных адресов ссылки и загрузки. Имена psect data в ОЗУ подобны **rdata\_0**, те что в ПЗУ похожи на **idata\_0**. Цифра относится к номеру банка ОЗУ.

Адреса ссылки и загрузки в отчете для psect, которые содержат объекты типа **bit**, имеют несколько иной смысл, чем обычные адреса ссылки и загрузки. В файле распределения памяти адрес ссылки является адресом ссылки psect, определенной как адрес bit. Адрес загрузки - адрес ссылки, определенный как адрес байта. Битовые объекты не могут быть инициализированы, поэтому отдельные адреса ссылки и загрузки не требуются. Компоновщик знает, как обработать эти psect по-другому из-за флага psect **BIT**. В файле распределения памяти psect bit обозначается значением масштаба 8. Это относится к числу объектов, которые могут быть расположены в адресуемой единице.

#### Классы psect

Теперь предположим, что в нашу систему добавлено дополнительное ОЗУ, так как программисты трудились и заполнили доступные в настоящее время 16 Кбайт. Несколько периферийных устройств были помещены в область от B000h до BFFFh, таким образом, дополнительное ПЗУ добавлено ниже нее, от 7000h до AFFFh. Теперь есть две отдельных области ПЗУ, и мы больше не можем давать единственный адрес для psect **text**.

Теперь мы можем воспользоваться этой дополнительной памятью определяя несколько диапазонов адресов, которые могут быть использованы для psect, размещаемых в ПЗУ. Это может быть сделано, создав класс psect. Есть несколько способов, с помощью которых psect могут быть связаны при использовании классов. Классы обычно используются компиляторами HI-TECH C, чтобы расположить код или секции psect **text**. Различные стратегии используются разными компиляторами, чтобы лучше соответствовать архитектуре процессора, для которой выполняется компиляция. Ниже рассматриваются некоторые из этих схем. Если вы намерены изменить опции компоновщика по умолчанию или создать свои собственные секции psect, проверьте опции компоновщика и директивы psect, сгенерированные генератором кода для конкретного компилятора, который вы используете.

Класс может быть определен с использованием другой опции компоновщика. Например, чтобы использовать дополнительную память, добавленную в нашу систему, мы могли бы определить класс, используя опцию компоновщика:

**-ACODE=7000h-AFFFh,C000h-FFFFh**

Опция это **-A**, за которыми следует имя класса, а затем список диапазонов адресов, разделенных запятыми. Помните, что это опция для компоновщика, а не CLD. Вышеупомянутый пример определяет два диапазона адресов для класса под названием **CODE**.

Вот как драйверы для компиляторов 8051, 8051XA и Z80 определяют опции, передаваемые компоновщику для обработки кода psect. В большой модели 8051 определения psect для секций psect, которые содержат код, следующие:

**psect text,class=CODE**

Флаг **CLASS** psect определяет, что psect **text** является членом класса под названием **CODE**.

Если определена единственная область ПЗУ или не используется опция **-ROM** в CLD или выбрано **single ROM** (единственное ПЗУ) в диалоговом окне **ROM & RAM addresses** (Адреса ПЗУ и ОЗУ) в HPD, класс не определен, и psect соединяются, используя опцию **-p**, как мы делали выше. Наличие psect с классом, но не определение этого класса приемлемо, при условии, что есть опция **-p** для явного расположения psect после их группировки. Если отсутствуют определение класса и опция **-p**, используется адрес памяти по умолчанию, который почти наверняка будет неуместным.

Если несколько областей ПЗУ были определены при помощи опции **-ROMranges** в CLD или при помощи определения диапазонов адресов в диалоговом окно **ROM & RAM addresses** (после нажатия кнопки **multiple ROMs**) в HPD, класс определяется драйвером, используя опцию компоновщика **-A**, аналогичную показанной выше, и опция **-p** исключается из опций, переданных компоновщику.

Компилятор PIC работает немного по-другому, поскольку ему приходится бороться с несколькими страницами ПЗУ, которые довольно малы. Генератор кода PIC определяет psect, содержащие код следующим образом.

**psect text0,local,class=CODE,delta=2**

Значение **DELTA** касается емкости ПЗУ и не должно рассматриваться здесь. Секции psect помещены в класс **CODE**, но обратите внимание, что они сделаны локальными, используя флаг **LOCAL** директивы psect. Секции psect, сгенерированные из функций C, имеют свои уникальные имена, которые продолжаются: **text0**, **text1**, **text2** и т.д. Локальные psect не группируются по модулям, т.е. если имеются два модуля, каждый из которых содержит локальную psect с тем же именем, они рассматриваются как отдельные psect. Локальные psect не могут быть размещены, используя опцию компоновщика **-p**, поскольку может существовать несколько psect с одинаковым именем. Локальные psect должны быть сделаны членами класса, и класс определен с помощью опции компоновщика **-A**. PIC работает таким образом, чтобы помочь с размещением кода ПЗУ страниц. Этот вопрос более подробно обсуждается в [Разделе 2.3.4](#_Проблемы_при_компоновке) на странице 32.

При использовании классов применяются несколько общих правил: если, например, вы хотите поместить psect, которая еще не входит в класс, в память, занятую классом, вы можете заменить адрес или имя psect в опции компоновщика **-p** с именем класса. Например, в типовом примере, рассмотренном выше, psect **const** помещалась в памяти после psect **text**. Если вы хотите, чтобы эта psect была помещена в память, назначенную классу **CODE**, можно использовать следующие параметры компоновщика.

**-pconst=CODE**

**-pvectors=0FFC0h**

**-pbss=0h,data/CODE**

**-ACODE=7000h-AFFFh,C000h-FFFFh**

Обратите также внимание, что место загрузки psect **data** было заменено после конца psect **const** в пределах памяти, назначенной классу **CODE**, чтобы проиллюстрировать, что адрес загрузки может быть указан с использованием имени класса.

Другое определение класса, которое иногда встречается в опциях компоновщика PIC, определяет три адреса для каждого диапазона памяти. Такая опция может выглядеть следующим образом:

**-AENTRY=0h-FFh-1FFh**

Первый диапазон определяет диапазон адресов, в котором должна начинаться psect. Секциям psect позволено продолжаться за второй адрес, пока они не выходят за последний адрес. Для примера выше, все секции psect, которые принадлежат классу ENTRY, должны начинаться в адресах между 0 и FFh. Секции psect должны закончиться перед адресом 1FFh. Никакая psect не может быть расположена так, чтобы ее начальный адрес находился между 100h и 1FFh. Компоновщик может, например, расположить две секции psect в этом диапазоне: первая охватывает адреса от 0 до 4Fh и вторая начинается в 50h и оканчивается в 138h. Такие опции компоновщика полезны для некоторых процессоров PIC (обычно базовый PIC) для кода psect, которые должны быть доступны для инструкций, изменяющих счетчик команд. Эти инструкции могут получить доступ только к первой половине каждой страницы ПЗУ.

#### Psect определяемые пользователем

Предположим теперь, что программист хочет включить специальный инициализированный объект C, который должен быть помещен в памяти по определенному адресу, то есть он не может быть просто помещен в psect data и связан с ним. В отдельном исходном файле программист помещает следующий код.

**#pragma psect data=lut**

**int lookuptable[] = {0, 2, 4, 7, 10, 13, 17, 21, 25};**

Директива #pragma в основном говорит, начиная с этого момента в этом модуле, все, что обычно входит в psect **data**, должно быть помещено в новую psect под названием **lut**. Поскольку массив инициализирован, он, как правило, будет помещен в data, и поэтому он будет перенаправлен в новую psect. Секция psect **lut** наследует любые параметры psect (определяемые флагами директивы psect), которые применяются к **data**.

Массив должен быть расположен в ОЗУ по адресу 500h. Опция **-p** выше могла быть изменена, чтобы включать также этот psect.

**-pbss=0h,data/const,lut=500h/**

(Адресу загрузки psect **data** были возвращены его предыдущие настройки.) Адреса для этой psect заданы как "500h/". "500h" адрес определяет адрес ссылки psect. Адрес загрузки может быть каким угодно, но желательно привязать его к существующей psect в ПЗУ. Если за адресом ссылки не следует адрес загрузки вообще, то адрес ссылки и загрузки будет тем же, в этом случае 500h. Символ "/", за которым не следует адрес, говорит компоновщику, что адрес загрузки должен располагаться сразу после конечного адреса загрузки предыдущей psect в опциях компоновщика. В этом случае это - адрес загрузки psect **data**, которая в свою очередь была помещена после psect **const**. Таким образом, в ПЗУ будут помещены psect **const**, **data** и **lut**, в перечисленном порядке.

Так как эта psect **data** - инициализируемая, она расположена в ПЗУ и должна быть скопирована в память, зарезервированную для нее в ОЗУ. Несмотря на то, что были определены различные адреса ссылки и загрузки с помощью опций компоновщика, программист должен будет предоставить код, который на самом деле выполняет копирование из ПЗУ в ОЗУ. (psect **data** создаваемая генератором кода обычно уже имеют код, предоставленный в файле времени выполнения для копирования psect.) Ниже приведен код C, который может выполнить копирование.

**extern unsigned char \*\_Llut, \*\_Hlut, \*\_Blut;**

**unsigned char \*i;**

**void copy\_my\_psect(void)**

**{**

**for(i=\_Llut; i<\_Hlut; i++, \_Blut++)**

**\*i = \*\_Blut;**

**}**

Обратите внимание, что, для получения доступа к символам **\_\_Llut** и т.д. из кода C, первый символ подчеркивания отбрасывается. Эти символы содержат адреса секций psect, таким образом, они объявлены (не определенными), поскольку указатель объектов в коде C, используют спецификатор extern. Помните, что объект **lookuptable** не будет инициализирован, пока эта функция C не будет вызвана и выполнена. Чтение из массива до инициализации, возвратит неправильные значения.

Если вы хотите инициализировать объекты, скопированные в ОЗУ до выполнения **main()**, вы можете написать ассемблерный код, или скопировать и изменить соответствующую подпрограмму в коде времени выполнения, предоставленную компилятором. Вы можете создать свой собственный объектный файл периода выполнения, предварительно скомпилировав измененный файл времени выполнения и используя этот объектный файл вместо стандартного файла, который автоматически соединяется с программами пользователя. При доступе к символам адреса psect на ассемблере требуются оба символа подчеркивания.

Если вы определяете свою собственную psect на основе psect **bss**, аналогичным образом вы должны предоставить код для очистки этой область памяти, если вы предполагаете, что объекты, определенные в psect, должны быть очищены перед использованием.

### Проблемы при компоновке

Компоновщик использует относительно сложный алгоритм для перемещения psect, содержащихся в передаваемых ему объектных и библиотечных файлах, но компоновщику требуется больше информации, чем описано выше, чтобы точно знать, как переместить каждую psect. Эта информация содержится в других опциях компоновщика, переданных компоновщику драйвером и флагами psect, которые используются каждой директивой psect. Ниже поясняются некоторые вопросы, которые должен учитывать компоновщик.

#### Страничная память

Предположим, что у процессора есть две области ПЗУ, в которые можно поместить код и постоянные данные. Компоновщик никогда не будет разделять psect ни по какой границе памяти. Граница памяти, как предполагается, существует везде, где есть разрыв в адресе, переданном компоновщику в опциях компоновщика. Например, если класс определен, используя адреса следующим образом:

**-ADATA=0h-FFh,100h-1FFh**

предполагается, что между адресом FFh и 100h существует некоторая граница, даже если эти адреса являются смежными. Именно поэтому вы будете видеть смежные диапазоны адресов, указанные таким образом, вместо одного диапазона, покрывающего всю область памяти. Чтобы упростить определение аналогичных смежных диапазонов адресов, можно использовать счетчик повторений, например может использоваться:

**-ADATA=0h-FFhx2**

В этом примере определены два диапазона: от 0 до FFh и затем от 100h до 1FFh. Некоторые процессоры имеют страницы памяти или банки. Опять же, psect не будет располагать границу между банком или страницей.

Учитывая, что секции psect не могут быть разделены по границам, наличие больших psect может быть проблемой для перемещения. Если имеются две области памяти по 1 Кбайт, и компоновщик должен разместить в этом пространстве одну psect размером 1,8 Кбайт, он не сможет выполнить это перемещение, даже если размер psect меньше, чем общий объем доступной памяти. Чем больше секции psect, тем труднее компоновщику разместить их. Если описанную выше psect разделить на три psect по 0.6 Кбайт, то компоновщик смог бы поместить две из них - по одной в каждой области памяти, но третья все равно не поместится в оставшееся пространство в обеих областях. При написании кода для процессоров, таких как PIC, которые помещают код сгенерированный из каждой функции C в отдельную локальную psect, функции не должны быть слишком длинными.

Если компоновщик не может расположить psect, он генерирует ошибку "Can't find space for psect xxxx" (Не удается найти место для psect xxxx), где xxxx - имя psect. Помните, что компоновщик перемещает секции psect, поэтому он не будет сообщать об ошибках памяти с указанием конкретных функций C или объектов данных. Изучите файл листинга ассемблера, чтобы идентифицировать, какая функция C связана с psect, упоминаемой в сообщении об ошибке, если локальные psect, созданы генератором кода.

Глобальные psect, которые не накладываются друг на друга, объединяются компоновщиком, для формирования одной psect перед перемещением. Есть случаи, кода эта сгруппированная psect, кажется, разделяются снова, чтобы поместить ее в память. Такие случаи происходят, когда класс psect, в котором он - участник, охватывает несколько диапазонов адресов, и сгруппированная psect слишком большая, чтобы соответствовать любому из диапазонов. Компоновщик может использовать промежуточные группировки psect, названные *сцепками* (clutches), для упрощения перемещения в диапазонах адресов класса. Сцепки никаким образом не управляются программистом, и для понимания и использования опций компоновщика полное понимание их природы не требуется. Достаточно сказать, что глобальные секции psect все еще могут использовать диапазоны адресов в пределах класса. Обратите внимание, что хотя сгруппированная psect может состоять из нескольких сцепок, индивидуальная psect, определенная в модуле, ни при каких обстоятельствах не может быть разделена.

#### Отдельные области памяти

Еще одна проблема, с которой сталкивается компоновщик, заключается в следующем: на некоторых процессорах существуют различные области памяти для программ и данных, т.е. микросхемах Гарвардской архитектуры как 8051XA. Например, ПЗУ может размещаться от 0h до FFFFh и отдельное ОЗУ может размещаться от 0h до 7FFh. Если компоновщику предлагают расположить psect по адресу 100h с помощью опции **-p**, как компоновщик узнает, является ли он адресом в памяти программы или в области данных? Компоновщик использует флаг **SPACE** директивы psect, чтобы определить это. Разным областям присваивается различное значение пространства. Например, для ПЗУ **SPACE** может быть присвоено значение 0 и для ОЗУ флаг **SPACE** 1. Флаги space для каждого psect приведены в файле распределения памяти.

Флаг space не используется, когда компоновщик может выделить область назначения объекта из адреса. Некоторые процессоры используют банки памяти, которые, с точки зрения процессора, охватывают тот же диапазон адресов, но которые находятся в одной и той же отдельной области памяти. В этих случаях компилятор присвоит уникальные адреса объектам в областях переключаемой памяти. Например, некоторые процессоры PIC могут получить доступ к четырем банкам ОЗУ, каждый из которых покрывает адреса от 0 до 7Fh. Компилятор присвоит объектам в первом банке (банк 0) адреса от 0 до 7Fh, объектам во втором банке: от 80h до FFh, объектам в третьем банке: от 100h до 17Fh и т.д. Эта дополнительная информация о банке удаляется из адреса, прежде чем он будет использоваться в ассемблерной инструкции. Все банки ОЗУ PIC используют флаг **SPACE** 1, но область ПЗУ на PIC совершенно отдельная и использует другой флаг **SPACE** (0). Флаг SPACE не относится к секциям psect, которые находятся в обеих областях памяти, таких как psect data, копируемые из ПЗУ в ПЗУ.

После завершения перемещения компоновщик будет соединять секции psect вместе, чтобы сформировать *сегмент* (segment). Сегменты, вместе с сцепками (clutches), редко упоминаются совместно с компилятором HI-TECH просто, потому что они являются абстрактными объектами, используемыми компоновщиком только во время его работы. Детали сегмента появятся в файле распределения памяти. Сегмент - набор секций psect, которые непрерывны и предназначены для определенной области в памяти. Имя сегмента происходит из имени первой psect, которое не следует путать с секцией psect, которая имеет это имя.

#### Объекты с абсолютными адресами

После перемещения psect адреса объектов данных могут быть разрешены и вставлены в инструкции ассемблера, которые ссылаются на адрес объекта. Существует одна ситуация, когда компоновщик не определяет и разрешает адрес объекта C. Это - когда объект был определен как абсолютный в коде C. Следующий пример показывает объект **DDRA**, располагаемый по адресу 200h.

**unsigned char DDRA @ 0x200;**

Когда генератор кода ссылается на объект **DDRA**, вместо того, чтобы использовать символ в сгенерированном ассемблерном коде, который будет позже заменен адресом объекта после перемещения psect, он сразу будет использовать значение 200h. Важно понять, что инструкции на ассемблере, которые обращаются к этому объекту, не будут иметь каких-либо символов, которые должны быть разрешены, и поэтому компоновщик просто пропустит их, поскольку они уже назначены. Если компоновщику было также сообщено с помощью его опций, что имеется память, доступная по адресу 200h для объектов ОЗУ, он может расположить psect, таким образом, что объект, который находится в этой psect также, использует адрес 200h. Поскольку символ, связанный с абсолютным объектом отсутствует, компоновщик не увидит, что два объекта совместно используют одну и ту же память. Если объекты наложатся, программа скорее всего будет работать непредсказуемо.

При расположении объектов по абсолютному адресу важно обеспечить, чтобы компоновщик не размещал объекты над теми, которые определены как абсолютные. Абсолютные объекты предназначены для объектов C, которые отображены поверх аппаратных регистров, чтобы облегчить доступ к регистрам из источника C. Программист должен убедиться в том, что параметры компоновщика не указывают, что в области памяти, занятой аппаратным обеспечением, имеется ОЗУ общего назначения. Обычные переменные, которые должны быть размещены по абсолютным адресам, должны быть сделаны с использованием отдельной psect (просто определяя свою собственную, используя директиву **psect** в ассемблерном коде или используя директиву **#pragma psect** в коде на C) и опцию компоновщика для размещения объектов. Если необходимо использовать абсолютный адрес для объекта в ОЗУ общего назначения, убедитесь, что опции компоновщика изменены таким образом, чтобы компоновщик не размещал другие разделы psect в этой области.

### Изменение параметров компоновщика

В большинстве случаях опции компоновщика по умолчанию не нуждаются в изменении. Если вы считаете, что опции должны быть изменены, но вы не понимаете, как они работают, рекомендуется, обратиться за технической помощью в отношении рассматриваемой проблемы.

Если нужно изменить параметры компоновщика, есть несколько способов сделать это. Если вы просто добавляете еще одну опцию к присутствующим по умолчанию, она может быть указана CLD, используя опцию **-L**. Чтобы расположить psect **lut**, которая использовалась в предыдущем примере, можно использовать следующую опцию.

**-L-plut=500/const**

Опция **-L** просто передает то, что следует компоновщику. Если вы хотите добавить другую опцию к опциям компоновщика по умолчанию, и вы используете HPD и проект, то это уже - простой случай открытия диалогового окна **Linker options...** (Опции компоновщика...) и добавление опции в конце присутствующих там. Опции должны быть введены точно, так как они должны быть представлены компоновщику, т.е. вам не нужен префикс **-L** в начале.

Если необходимо изменить существующие опции компоновщика за исключением простого изменения адреса памяти, которые определены с помощью опции **-A** CLD, то вы не можете использовать CLD, чтобы сделать это непосредственно. Вы должны выполнить компиляцию и компоновку отдельно. Допустим, что файл **main.c** и **extra.c** должны быть скомпилированы для процессора 8051 с измененными опциями компоновщика. Сначала мы можем выполнить компиляцию, но не включать этап компоновки при помощи командной строки вроде этой.

**c51 -o -Zg -asmlist -C main.c extra.c**

Опции **-C** останавливают компиляцию перед этапом компоновки. Включите любые другие опции, которые обычно требуются. Это создаст два объектных файла: **main.obj** и **extra.obj**, которые затем должны быть соединены.

Снова выполните команду CLD в многословном режиме, задав опцию **-v** в командной строке, передав ей имена объектных файлов, созданных выше, и перенаправьте вывод в файл. Например:

**c51 -v -A8000,0,100,0,0 main.obj extra.obj > main.lnk**

Обратите внимание, что, если вы не дадите CLD опцию **-A**, компилятор предложит вам ввести адреса памяти, но с перенаправленным выводом, вы не будете видеть подсказки.

Сгенерированный (**main.lnk**) файл будет содержать командную строку, которую сгенерировал CLD, чтобы выполнить компоновщик со значениями памяти, определенными, используя опцию **-A**. Отредактируйте этот файл и удалите любые сообщения, распечатанные компилятором. Удалите командную строку для любых приложений, запущенных за этапом компоновки, например **objtohex** или **cromwell**, хотя вы должны принять во внимание эти команды, так как вам нужно будет запускать эти приложения вручную после этапа компоновки. Командная строка компоновщика обычно очень длинная и если используется пакетный файл DOS для выполнения этапа компоновки, он ограничен строками 128 символов в длину. Вместо этого компоновщику может быть передан командный файл, который содержит только опции компоновщика. Разбейте командную строку компоновщика в файле, который вы создали, вставив символы наклонной черты влево "\" сопровождаемые Return. Также удалите имя и путь исполнимой программы компоновщика в начале командной строки так, чтобы остались только опции. В приведенной выше командной строке сгенерирован файл **main.lnk**, который затем был отредактирован, как предложено выше, чтобы дать следующее.

**-z -pvectors=08000h,text,code,data,const,strings \**

**-prbit=0/20h,rbss,rdata/strings,irdata,idata/rbss \**

**-pbss=0100h/idata -pnvram=bss,heap -ol.obj \**

**-m/tmp/06206eaa /usr/hitech/lib/rt51--ns.obj main.obj \**

**extra.obj /usr/hitech/lib/51--nsc.lib**

Теперь, с осторожностью, измените опции компоновщика в этом файле в соответствии с вашими требованиями.

Затем выполните этап компоновки, запустив непосредственно компоновщик и перенаправив его опции из созданного командного файла.

**hlink < main.lnk**

Это создаст выходной файл с именем **l.obj**. Если после этапа компоновки были запущены другие приложения, вы должны выполнить их, чтобы создать выходной файл с корректным форматом, например HEX файл.

Изменение опций в HPD гораздо проще. Снова, просто откройте диалоговое окно **Linker options...** (Опции компоновщика...) и внесите требуемые изменения, используя кнопки в нижней части окна, чтобы помочь с редактированием. Сохраните и повторно соберите свой проект.

Файл распределения памяти (map) будет содержать командную строку, реально переданную компоновщику, и ее можно проверить, чтобы убедиться, что компоновщик работал с новыми опциями.

# Использование HPDZ

## Введение

В этой главе описывается интегрированная среда системы разработки программиста (HPD[[11]](#footnote-11)) HI-TECH C. Предполагается, что вы уже установили HI-TECH C компилятор. Если компилятор еще не установлен, перейдите к [Главе 1 Введение](#_Введение_2) и следуйте инструкциям по установке. HPDZ является версией HPD, применимой к компилятору Z80.

### Запуск HPDZ

Чтобы запустить HPDZ, просто введите **HPDZ** в командной строке DOS. После краткого периода активности диска вы увидите экран, аналогичный показанному на [Рисунке 3-1](#r31) на странице 37.

Рисунок 3‑1 Экранная заставка HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:55:53 AM.

**┌─ untitled ───────────────────────────────────────────────────────────────────┐**

**│\_ ↑**

**│ ◀**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│Line 1/1 Col 1 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180 V7.60 ─────────────────────────────────────┐**

**│** HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/180/64180 V7.60)  **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to:  **│**

**│** Clyde  **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

Начальный экран HPDZ разбит на три окна. Верхнее окно содержит *строку меню*, среднее окно - *текстовый редактор* HPDZ, а нижнее окно - *окно сообщений*. При выборе некоторых пунктов меню могут появиться другие окна. Окно редактора вы будете использовать большую часть времени.

HPDZ использует оконный пользовательский интерфейс HI-TECH для обеспечения текстового экранного интерфейса пользователя. Он имеет несколько перекрывающихся окон и выпадающих меню. Далее в этой главе описаны функции пользовательского интерфейса, общие для всех оконных приложений HI-TECH,.

Кроме того, HPDZ может использовать один параметр командной строки. Это либо имя текстового файла, либо имя файла проекта. (Файлы проекта обсуждаются в следующем разделе этой главы). Если аргумент имеет расширение .PRJ, HPDZ попытается загрузить файл проекта с этим именем. Имена файлов с любым другим расширением будут обрабатываться как текстовые файлы и загружены в редактор.

Если будет задан параметр без расширения, HPDZ сначала попытается загрузить файл .PRJ, затем файл .C. Например, если текущий каталог будет содержать файл с именем X.C, и HPDZ вызван командой HPDZ X, он сначала попытается загрузить X.PRJ и когда это не получится, загрузит X.C в редактор. Если исходный файл не загружен в редактор, будет запущен пустой файл с именем "untitled" (неназванный).

## Оконный интерфейс пользователя HI-TECH

Оконный пользовательский интерфейс HI-TECH, используемый HPDZ, обеспечивает мощный текстовый экранный интерфейс пользователя. С ним можно общаться как с помощью клавиатуры, так и в комбинации клавиатуры с мышью. Для новых пользователей большинство операций будет проще выполнять с помощью мыши, однако по мере накопления опыта работы с пакетом вы узнаете последовательности горячих клавиш для наиболее часто используемых функций.

### Требования к оборудованию

Оконные приложения HI-TECH будут работать на любом MS-DOS совместимом компьютере со стандартной видеокартой, поддерживающей текстовые режимы экрана с 80 столбцами и 25 строками или более. Текстовые режимы с большим разрешением, как режим EGA 80 x 43 будут распознаваться и использоваться, если режим был выбран ранее, до выполнения HPDZ. Режимы с более высоким разрешением могут использоваться также с опцией /SCREEN:XX, как описано ниже. С некоторыми плохо написанными утилитами VGA могут возникнуть проблемы. Они могут инициализировать аппаратные средства в режим с более высоким разрешением, но оставить области данных BIOS с низким объемом памяти, установив значения отображения 80 x 25.

HPDZ может устанавливать режим отображения экрана дисплеев EGA или VGA для отображения более 25 строк. Опция /SCREEN:nn, где nn является одним из значений 25, 28, 43 или 50, заставит HPDZ устанавливать отображение в это или максимально близкое количество строк. Дисплей EGA поддерживает отображение только для 25 и 43 строк, в то время как VGA также поддерживает 28 и 50 строк.

Отображение вернется предыдущему режиму после выхода из HPDZ. Выбранное количество строк будет сохранено в файле HPDZ.INI и использоваться при последующих вызовах HPDZ, если не будет переопределено другой опцией /SCREEN.

HPDZ будет распознавать и использовать любой драйвер мыши, который поддерживает стандартный интерфейс Int 33H. Почти все современные драйверы мыши поддерживают стандартный интерфейс драйвера устройства. В некоторых старых драйверах мыши отсутствует ряд вызовов состояния драйвера. Если вы будете использовать такой драйвер мыши, HPDZ будет по прежнему работать с мышью, но диалоговое окно установки мыши в меню **«»** работать не будет.

### Цвета

Цвета используются в HPDZ двумя способами. Во-первых, есть цвета, связанные с отображением экрана. Их можно изменить в соответствии с вашими предпочтениями. Второе использование цвета - это необязательное кодирование текста, введенного в текстовое окно. Это поможет вам увидеть различные элементы программы по мере ее ввода и компиляции. Эти цвета также можно изменить в соответствии с вашими требованиями. Цвета включают два элемента, фактический цвет и его атрибуты, такие как яркий или инверсный. В [Таблице 3-1](#t31) на странице 39 показаны цвета и их значения, а в [Таблице 3-2](#t32) на странице 39 показаны атрибуты и их значения.

Таблица 3‑1 Значения цветов

|  |  |
| --- | --- |
| **Значение** | **Цвет** |
| **0** | черный |
| **1** | голубой |
| **2** | зеленый |
| **3** | синий |
| **4** | красный |
| **5** | малиновый |
| **6** | коричневый |
| **7** | белый |
| **8** | серый |
| **9** | светло-голубой |
| **10** | светло-зеленый |
| **11** | светло-синий |
| **12** | светло-красный |
| **13** | светло-малиновый |
| **14** | желтый |
| **15** | ярко-белый |

Для переднего плана допустимы любые цвета, но для фона можно использовать только цвета от 0 до 7. В [Таблице 3-3](#t33) на странице 40 показаны настройки определения цветов, используемых редактором при выборе цветового кодирования (подсветки синтаксиса).

Таблица 3‑2 Цвета атрибутов

|  |  |
| --- | --- |
| **Атрибут** | **Описание** |
| **normal:** | Цвет обычного текста |
| **bright:** | Яркий / выделенный цвет текста |
| **inverse:** | Цвет инверсного текста |
| **frame:** | Цвет границы окна |
| **title:** | Цвет заголовка окна |
| **button:** | Цвет любой кнопки в окне |

Стандартные цветовые схемы для цветов дисплея и цветового кодирования текстового редактора можно увидеть в разделе настроек цвета в файле HPDZ.INI. Первое значение в определении цвета обозначает цвет переднего плана, а второе соответствует цвету фона. Чтобы задать цвета, отличные от наборов по умолчанию, вы должны удалить # перед каждой строкой, а затем выбрать новое значение цвета.

Таблица 3‑3 Настройки цветов кода

|  |  |
| --- | --- |
| **Настройки** | **Описание** |
| **C\_wspace:** | Пустое пространство - цвет переднего плана влияет на курсор |
| **C\_number:** | Восьмеричные, десятичные и шестнадцатеричные числа |
| **C\_alpha:** | Имена алфавитно-цифровых переменных, макросов и функций |
| **C\_punct:** | Символы пунктуации и т.д. |
| **C\_keyword:** | Ключевые слова C и типы переменных: например, int, static и т.д. |
| **C\_brace:** | Открывающие и закрывающие фигурные скобки: {} |
| **C\_s\_quote:** | Текст в одинарных кавычках |
| **C\_d\_quote:** | Текст в двойных кавычках |
| **C\_comment:** | Комментарии C традиционного стиля: /\*...\*/ |
| **Cpp\_comment** | Комментарии в стиле C++: //... |
| **C\_preprocessor:** | Директивы препроцессора C: #blah |
| **Include\_file:** | Имена включаемых файлов |
| **Error:** | Ошибки - что-либо неправильное обнаруженное редактором |
| **Asm\_code:** | Встроенный ассемблерный код (#asm...#endasm) |
| **Asm-comment:** | Комментарии ассемблера: ;... |

Файл .INI также содержит пример альтернативы стандартной цветовой схемы. Тот же процесс может использоваться для настройки цветовой схемы для строк меню и меню.

### Выпадающее меню

Окна HI-TECH включает в себя систему выпадающих меню, которые раскрываются из строки меню в верхней части экрана. Строка меню разбивается на ряд слов или символов, каждый из которых является названием отдельного выпадающего меню.

Система меню может использоваться с клавиатурой, мышью или комбинацией действий клавиатуры и мыши. Поддерживаемые действия клавиатура и мыши перечислены на странице 40 в [Таблице 3-4](#t34).

Таблица 3‑4 Клавиши и действия мыши системного меню

|  |  |  |
| --- | --- | --- |
| **Действие** | **Клавиша** | **Мышь** |
| **Открытие меню** | Alt-пробел | Нажатие левой кнопки в строке меню или нажатие средней кнопки в любом месте экрана |
| **Выход из меню** | Alt-пробел или Escape | Нажатие левой кнопки на отображении системы меню |
| **Выбор элемента** | Enter | Отпускание левой или центральной кнопки выделенного элемента или нажатие левой или центральной кнопки на элементе |
| **Следующее меню** | Стрелка вправо | Перетаскивание вправо |
| **Предыдущее меню** | Стрелка влево | Перетаскивание влево |
| **Следующий элемент** | Стрелка вниз | Перетаскивание вниз |
| **Предыдущий элемент** | Стрелка вверх | Перетаскивание вверх |

#### Выбор меню с помощью клавиатуры

Для выбора пункта меню с клавиатуры нажмите **alt-Space**, чтобы открыть систему меню. Затем используйте клавиши со стрелками, чтобы переместиться в желаемое меню и выделить требуемый пункт. Когда требуемый пункт выделен выберите его, нажав **Enter**. Некоторые пункты меню будут выведены на экран с более низкой интенсивностью или другим цветом и не могут быть выбраны. Эти элементы отключены, потому что их выбор не уместен в текущем контексте приложения. Например, элемент **Save project** (Сохранение проекта) нельзя выбрать, если проект не был загружен или определен.

#### Выбор меню с помощью мыши

Выбор пункта меню с помощью мыши, новым пользователям кажется несколько неудобным. Однако, вскоре становится второй натурой. Для открытия системного меню, наведите курсор мыши на название требуемого меню и нажмите левую кнопку. Вы можете просматривать систему меню, удерживая левую кнопку и перетаскивая указатель мыши через названия нескольких меню, открывая каждое поочередно. Вы можете также управлять системой меню со средней кнопкой на трехкнопочных мышах. Нажмите среднюю кнопку, чтобы открыть строку меню. Это делает возможным ее выбор, даже если она полностью скрыта масштабируемым окном.

После открытия меню, возможны два способа выбора. Если левая или средняя кнопка будут отпущены, пока пункт меню не выделен, меню останется открытым. Тогда вы можете выбрать используя клавиатуру или переместив указатель на желаемый пункт меню и щелкнув по левой или средней кнопке мыши. Если после открытия меню кнопка мыши остается нажатой, вы можете осуществить выбор, переместив мышь к нужному пункту и отпустив кнопку.

#### Горячие клавиши меню

При просмотре системы меню вы заметите, что в некоторых пунктах меню отображаются последовательности горячих клавиш. Например, пункт меню HPDZ Save (Сохранить), содержит последовательность нажатий клавиш **alt-S**. Когда пункт меню имеет клавиатурный эквивалент, он может быть выбран непосредственно, нажав эту клавишу, не открывая систему меню. Клавиатурные эквиваленты будут или клавишами **alt-алфавитно-цифровые** клавишами или функциональными клавишами. Где используются функциональные клавиши, обычно различные, но связанные пункты меню будут группироваться на одной клавише. Например, в HPDZ **F3** присвоена для Compile and Link (Скомпилировать и скомпоновать), **shift-F3** присвоена Compile to .OBJ (Скомпилировать в .OBJ), и **ctrl-F3** присвоена Compile to .AS (Скомпилировать в .AS).

Клавиатурные эквиваленты также присвоены всем меню, обеспечивая удобный способ перехода в конкретное меню с помощью одного нажатия клавиши. Присвоенной клавишей обычно будет **alt** и первая буква имени меню, например **alt-E** для меню Edit. Клавиатурные эквиваленты меню выделены другим цветом (кроме монохромных дисплеев) и отображаются в инверсном виде, когда нажата клавиша **alt**. Полный список клавиатурных эквивалентов HPDZ показан в [Таблице 3-5](#t35) на странице 42.

### Выбор окон

Окна HI-TECH позволяют накладывать или разбивать окна. С помощью клавиатуры, вы можете перенести окно на передний план, нажав ctrl-Enter один или несколько раз. Каждый раз, когда нажимается ctrl-Enter, заднее окно переносится вперед, а каждое другое окно на экране перемещается на один уровень назад. Серия нажатий ctrl-Enter будет бесконечно чередовать иерархию окон.

Таблица 3‑5 Горячие клавиши вызова меню HPDZ

| **Клавиши** | **Назначение** |
| --- | --- |
| **Alt-O** | Открытие файла для редактирования |
| **Alt-N** | Очистка редактируемого файла |
| **Alt-S** | Сохранение редактируемого файла |
| **Alt-A** | Сохранение редактируемого файла под другим именем |
| **Alt-Q** | Выход в DOS |
| **Alt-J** | Интерпретатор команд DOS |
| **Alt-F** | Открытие меню файл |
| **Alt-E** | Открытие меню редактирования |
| **Alt-I** | Открытие меню компиляции |
| **Alt-M** | Открытие меню создания |
| **Alt-R** | Открытие меню запуска |
| **Alt-T** | Открытие меню опций |
| **Alt-U** | Открытие меню утилит |
| **Alt-H** | Открытие меню помощи |
| **Alt-P** | Открытие файла проекта |
| **Alt-W** | Диалог уровня предупреждений |
| **Alt-Z** | Меню оптимизации |
| **Alt-D** | Command.com |
| **F3** | Компиляция и сборка одного файла |
| **Shift-F3** | Компиляция в объектный файл |
| **Ctrl-F3** | Компиляция в ассемблерный код |
| **Ctrl-F4** | Получение последнего файла |
| **F5** | Создание целевой программы |
| **Shift-F5** | Повторная компоновка целевой программы |
| **Ctrl-F5** | Повторное создание всех объектных и целевых файлов |
| **Alt-P** | Загрузка файла проекта |
| **Shift-F7** | 1 команда определенная пользователем |
| **Shift-F8** | 2 команда определенная пользователем |
| **Shift-F9** | 3 команда определенная пользователем |
| **Shift-F10** | 4 команда определенная пользователем |
| **F2** | Поиск в окне редактирования |
| **Alt-X** | Вырезать в буфер обмена |
| **Alt-C** | Скопировать в буфер обмена |
| **Alt-V** | Вставить из буфера обмена |

С помощью мыши, вы можете переместить любое видимое окно на передний план, нажав левую кнопку в области его содержимого[[12]](#footnote-12). Окно может быть сделано последним, удержав клавишу **alt** и нажав левую кнопку в области его содержимого. Если окно полностью скрыто другими окнами, обычно оно может располагаться или нажатием **ctrl-Enter** несколько раз или перемещая другие окна назад с помощью **alt-левая кнопка**.

Некоторые окна не будут выводиться на передний план, когда в них нажимается левая кнопка. Эти окна имеют специальный атрибут, установленный приложением, и обычно делаются такими по уважительной причине. Чтобы привести пример, чаще всего, окно ошибок компилятора HPDZ не будет выводится на передний план если щелкнуть по нему. Вместо этого он воспримет клик, как если бы оно было уже окном переднего плана. Это позволяет использовать мышь для выбора перечисленных ошибок компилятора, оставляя спереди окно редактора, таким образом, позволяя изменять текст программы.

### Перемещение и изменение размеров окон

Большинство окон может быть перемещено и изменено пользователем. На экране отсутствуют признаки, позволяющие отличить окна, которые не могут быть перемещены или изменены. Если при попытке переместить или изменить размеры окна ничего не происходит, это указывает, что окно не может быть изменено. Некоторые окна могут быть перемещены, но не изменены, обычно потому, что их содержание имеет фиксированный размер и изменение размеров не имеет смысла. Калькулятор HPDZ - пример окна, которое может быть перемещено, но не изменено.

Окна могут быть перемещены и изменены в размере используя клавиатуру или мышь. Используя клавиатуру, режим перемещения/изменения может быть введен нажатием **ctrl-alt-пробел**. Приложение ответит, заменяя строку меню перемещения/изменения на ленточное меню. Она позволяет перемещать и изменять большую часть окон переднего плана. Когда изменение размеров завершено, вы должны нажать **Enter**, чтобы вернуться к операционной функции окна. В [Таблице 3-6](#t36) на странице 43приведенполный список всех операционных клавиш.

Таблица 3‑6 Клавиши режима изменения размера

|  |  |
| --- | --- |
| **Клавиша** | **Действие** |
| **Стрелка влево** | Переместить окно вправо |
| **Стрелка вправо** | Переместить окно влево |
| **Стрелка вверх** | Переместить окно вверх |
| **Стрелка вниз** | Переместить окно вниз |
| **Shift-Стрелка влево** | Сжать окно горизонтально |
| **Shift-Стрелка вправо** | Развернуть окно по горизонтали |
| **Shift-Стрелка вверх** | Вертикальное укорачивание окна |
| **Shift-Стрелка вниз** | Развернуть окно по вертикали |
| **Enter или Escape** | Выход из режима перемещения/изменения |

Режим перемещения/изменения может быть закрыт любым обычным действием приложения, например, щелчком мыши, нажатием горячей клавиши или активацией системы меню нажатием **alt-пробел**. Существуют другие способы перемещения и изменения размера окон:

* Окна можно перемещать и масштабировать с помощью мыши. Вы можете переместить любое видимое окно, нажав левую кнопку мыши на ее рамке, перетащив его в новое место и отпустив кнопку. Если окно будет "захвачено" около одного из его углов, то указатель изменится на ромб. Тогда вы можете перемещать окно в любом направлении, в том числе по диагонали. Если окно будет захвачено в середине верхнего или нижнего края, то указатель изменится на вертикальную стрелку. Тогда вы можете переместить окно вертикально. Если окно будет захвачено около середины левого или правого края, то указатель изменится на горизонтальную стрелку. Тогда окно можно переместить только по горизонтали.
* Если в окне есть *полоса прокрутки*, нажатие левой кнопки мыши на полосе прокрутки не перемещает окно. Вместо этого он активирует полосу прокрутки, отправляя сообщения прокрутки в приложение. Если вы хотите переместить окно, у которого есть рамка с полосой прокрутки, просто выберите другую часть рамки.
* Окна могут быть изменены с помощью правой кнопки мыши. Вы можете изменить любое видимое окно нажатием правой кнопки мыши на нижний или левой рамке. Затем перетащите рамку к новой границе и отпустите кнопку. Если окно захвачено около его правого нижнего угла указатель изменится на ромб, и можно изменить размеры окна в любом направлении. Если рамка захвачена где-нибудь еще на нижней кромке, размеры могут изменяться только вертикально. Если окно захвачено где-нибудь еще с правого края, размеры можно изменить только горизонтально. Если правая кнопка будет нажата в любом месте верхнего или левого края, то ничего не произойдет.
* Вы также можете *масштабировать* (zoom) окно до максимального размера. Большая часть окон переднего плана могут масштабироваться, нажатием shift-(на блок дополнительных клавиш)+, если оно масштабируется снова, то возвращается к своему прежнему размеру. В масштабируемом или не масштабированном состоянии окно может быть перемещено и изменено. Масштабирование эффективно переключает между двумя определяемыми пользователем размерами. Вы можете также увеличить окно, щелкая правой кнопкой мыши в области его содержания.

### Кнопки

Некоторые окна содержат кнопки, которые могут использоваться, чтобы сразу выбрать определенные действия. Кнопки подобны пунктам меню, которые всегда видимы и могут быть выбраны. Кнопка может быть нажата или щелкнув левой кнопкой мыши по ней или при помощи ее клавиатурного эквивалента. Клавиатурный эквивалент кнопки отображается как часть кнопки, или как часть поясняющего сообщения где-то в другом месте окна. Например, окно ошибок HPDZ ([Рисунок 3-6](#r36) на странице 49) содержит много кнопок, для выбора HELP, необходимо щелкнуть левой кнопкой мыши по нему или нажать **F1**.

### Меню настроек

При открытии системного меню, обозначенного символом **«»** в строке меню, вы найдете две записи: запись *About HPDZ* (о HPDZ), которая отображает информацию о номере версии HPDZ и запись *Setup* (Настройки). Выбор записи Setup открывает диалоговое окно, показанное на [Рисунке 3-2](#r32) на странице 45. Это диалоговое окно отображает информацию об использовании памяти HPDZ, и позволяет устанавливать чувствительность мыши, отображение времени суток в строке меню и использование звука. После изменения параметров чувствительности мыши можно протестировать их, нажав на кнопку Test (Проверка). Это изменит значения мыши, таким образом, вы сможете протестировать измененную чувствительность. Если вы впоследствии нажмете Cancel, они вернутся к предыдущим значениям. Выбор OK подтвердит измененные значения и сохранит их в файле инициализации HPDZ, таким образом, они будут загружены в следующий раз, при запуске HPDZ. Настройки звука и часов будут сохранены в файле инициализации при выборе OK.

Рисунок 3‑2 Диалог настройки

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:55:53 AM.

**┌─ untitled ───────────────────────────────────────────────────────────────────┐**

**│ ↑**

**│ ◀**

**│ .**

**│ .**

**│ ╔═ Setup ═════════════════════════════════════════════════╗ .**

**│ ║┌─ Mouse setup ────────────────────────────────────────┐ ║ .**

**│ ║│ Sensitivity, vertical: 50. . Horizontal: 50. . │ ║ .**

**│ ║│ │ ║ .**

**│ ║│ Ballistic threshold: 50. . Click repeat rate: 9. │ ║ .**

**│ ║└──────────────────────────────────────────────────────┘ ║ .**

**│ ║ ║ .**

**│ ║ [X] Enable sound - F3 [X] Enable clock – F4 ║ .**

**│ ║ ║ .**

**│ ║ Mouse driver v8.00, type = InPort mouse, IRQ = 0x02 ║ .**

**│ ║ Display type: 80x28, VGA with analog color monitor ║ .**

**│ ║ 80386/486 cpu, DOS V5.00, 209120 bytes free ║ .**

**│ ║ Near heap sise = 6144, free stack space - 9584 ║ .**

**│ ║ Using XMS V2.00, 512K used, 398K free ║ ↓**

**└←▲.......║ ║░░░░░░░░░→┘**

**┌─ Project║ [ OK – Enter ] < Cancel – Esc > <Test - F2 > ║──────────┐**

**│** HI**╚═════════════════════════════════════════════════════════╝**60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

## Учебник: создание и компиляция программы на C

Этот учебник должен быть достаточным для того, чтобы вы начали использовать HPDZ. Он не пытается дать вам всестороннюю экскурсию по функциям HPDZ, которая находится в справочном разделе этой главы. Даже если вы опытный программист на C, но до этого не использовали оконное приложение HI-TECH, мы настоятельно рекомендуем завершить этот урок.

Прежде, чем запустить HPDZ, необходимо создать рабочий каталог. Удостоверьтесь, что вы зарегистрированы к корневом каталоге на жестком диске и введите следующие команды:

C:\> **md tutorial**

C:\> **cd tutorial**

C:\TUTORIAL> **HPDZ**

Вам будет представлен экран запуска HPDZ. На этом этапе редактор готов принять любой набираемый вами текст. Мигающий курсор блока виден в верхнем левом углу окна редактирования. Теперь вы готовы ввести свою первую программу на C, используя HPDZ. Естественно это будет пресловутая программа "Hello world" (Привет мир).

Введите следующий текст, нажав Enter один раз в конце каждой строки. Вы можете ввести пустые строки, нажав Enter, не вводя текста.

**#include <stdio.h>**

**main()**

**{**

**printf("Hello, world")**

**}**

Обратите внимание, что точка с запятой была сознательно опущена в конце оператора printf(), чтобы продемонстрировать средства обработки ошибок HPDZ. [Рисунок 3-3](#r33) на странице 46, показывает выглядит экран после ввода программы "Hello world").

Рисунок 3‑3 Программа Hello в HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:56:37 AM.

**┌─ C:\HELLO.C ─────────────────────────────────────────────────────────────────┐**

**│**#include <stdio.h> **↑**

**│ .**

**│**/\* The infamous hello world program. \*/ **.**

**│ .**

**│**void main(void) **.**

**│**{ **.**

**│** printf(("Hello, world")  **.**

**│ .**

**│**}**\_ .**

**│ .**

**│ .**

**│ ◀**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│Line 8/8 Col 2 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180 V7.60 ─────────────────────────────────────┐**

**│** HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/180/64180 V7.60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

Теперь имеется введенная программа C (с одной ошибкой!) и почти готовая к компиляции. Вы должны сохранить ее в файл на диске, и затем вызвать компилятор. Для сохранения исходного кода на диск, необходимо выбрать пункт **Save** из меню **File** ([Рисунок 3-4](#r34) на странице 47).

При отсутствии мыши, выполните следующие шаги:

* Откройте системное меню, нажав **alt**-**пробел**
* Перейдите к меню **Edit**, используя клавишу со стрелкой вправо
* Переместитесь вниз к пункту **Save**, используя клавишу со стрелкой вниз
* Когда пункт **Save** будет выделен, выберите его, нажав клавишу Enter.

Если вы используете мышь, выполняете эти шаги:

* Откройте меню **File**, переместив указатель на слово **File** в строке меню и нажав левую кнопку
* Выделите пункт **Save**, перетащив мышь вниз с левой удерживаемой кнопкой, пока пункт **Save** не будет выделен
* Когда пункт **Save** будет выделен, выберите его, отпустив левую кнопку.

Рисунок 3‑4 Меню File HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:57:18 AM.

**┌─ un├──────────────────────┤──────────────────────────────────────────────────┐**

**│ │.Open ... alt-O.│ ↑**

**│ │ New alt-N │ ◀**

**│ │ Save alt-S │ .**

**│ │ Save as ... alt-A │ .**

**│ │ Autosave ... │ .**

**│ ├──────────────────────┤ .**

**│ │ Quit alt-Q │ .**

**│ ├──────────────────────┤ .**

**│ │ Clear pick list │ .**

**│ ├──────────────────────┤ .**

**│ │ C:\HELLO.C ctrl-F4 │ .**

**│ └──────────────────────┘ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│Line 1/1 Col 1 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180 V7.60 ─────────────────────────────────────┐**

**│** HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/180/64180 V7.60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

В момент когда меню **File** ([Рисунок 3-4](#r34) на странице 47) открыто, вы можете заметить, что пункт **Save** включает текст **alt-S** на правом крае меню. Это означает, что доступ к команде сохранения также можно получить напрямую с помощью горячей клавиши **alt-S**. Ряд наиболее часто используемых команд меню имеет эквиваленты горячих клавиш, которые будут либо **alt-алфавитно-цифровыми** последовательностями, либо функциональными клавишами.

После того, как выбран пункт **Save**, вам будет предложено ввести имя файла. Если драйвер HPDZ нуждается в дополнительной информации, такой как имя файла, прежде чем он сможет выполнить команду, он всегда предложит вам стандартный диалог, подобный приведенному ниже.

Диалоговое окно содержит строку редактирования, в котором вы можете ввести имя файла, которое будет использоваться, и несколько кнопок. Они могут использоваться для выполнения различных действий в диалоговом окне. Кнопка может быть нажата, щелкнув левой кнопкой мыши с указателем, расположенным на ней, или при помощи ее клавиатурного эквивалента. Текст в строке редактирования может быть отредактирован, используя стандартные клавиши редактирования: стрелка влево, стрелка вправо, клавиша Backspace, Del и Ins. Клавиша Ins переключает строчный редактор между режимом вставки и перезаписи.

В этом случае сохраните свою программу C в файл, названный "hello.c". Введите **hello.c** и затем нажмите **Enter**. Должен быть небольшой период активности диска, пока HPDZ сохраняет файл.

Вы должны установить конфигурацию памяти. Для этого выберите пункт **ROM & RAM addresses...** (Адреса ПЗУ и ОЗУ...) из меню **Options** (Опции). Вы должны использовать 0 для адреса ПЗУ, 8000 для адреса ОЗУ и 8000 для размера ОЗУ. Если вы не используете энергонезависимое ОЗУ, или все ваше ОЗУ является энергонезависимым - установите NVRAM равным 0. После того как вы установили настройки памяти диалоговое окно должно выглядеть как на [Рисунке 3-5](#r35) на странице 48.

Рисунок 3‑5 Диалоговое окно адреса ПЗУ и ОЗУ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:57:57 AM.

**┌─ untitled ───────────────────────────────────────────────────────────────────┐**

**│ ↑**

**│ ◀**

**│ ╔═ Enter ROM and RAM addresses in hex: ══════════════════════════════════╗ .**

**│ ║ ║ .**

**│ ║ ┌─ ROM and RAM addresses ─┐ ║ .**

**│ ║ │ ROM addresses: 0 . │ ║ .**

**│ ║ │ │ ║ .**

**│ ║ │ RAM addresses: 8000. │ ║ .**

**│ ║ │ │ ║ .**

**│ ║ │ RAM size: 8000. │ ║ .**

**│ ║ │ │ ║ .**

**│ ║ │ RAM addresses: 0 . │ ║ .**

**│ ║ └─────────────────────────┘ [X] Initialised data in RAM – F3 ║ .**

**│ ║ ║ .**

**│ ║ ║ .**

**│ ║ ║ .**

**│ ║ ║ .**

**│Li║ ║ ↓**

**└←▲║ ║░→┘**

**┌─ ║ [ OK – Enter ] < Cancel – Esc > < Help – F1 > < Multiple ROMs – F2 > ║──┐**

**│ ║ ║ │**

**│ ╚════════════════════════════════════════════════════════════════════════╝ │**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde  **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

Теперь вы готовы к компиляции программы. Для компиляции и компоновки в один шаг, выберите пункт **Compile and link** (Компиляция и компоновка) из меню **Compile** (Компиляция), используя выпадающее системное меню, как раньше. Обратите внимание, что пункту **Compile and link** присвоена клавиша **F3**: в будущем, вы можете сэкономить время с помощью этой клавиши.

На этот раз компилятор не будет работать до завершения. Это происходит потому, что мы сознательно опустили точку с запятой в конце строки, чтобы увидеть, как HPDZ обрабатывает ошибки компилятора. Через пару секунд дисковой активности, пока выполняются фазы компилятора CPP и P1, Вы должны услышать звук "привлечения внимания". Окно сообщения будет заменено окном, содержащим много кнопок и сообщение "; expected" (ожидается ;), как показано в [Рисунке 3-6](#r36) на странице 49.

Текст в рамке окна ошибок показывает количество сгенерированных ошибок компилятором, и какая фаза компилятора сгенерировала их. Большинство ошибок возникают в P1.EXE и CGxx.EXE. Также ошибки могут возвращать CPP.EXE и LINK.EXE. В данном случае рамка окна ошибок содержит сообщение "1 error, 0 warnings from p1.exe" (1 ошибка, 0 предупреждений в p1.exe) указывающее что проход 1 компилятора нашел 1 неустранимую ошибку. Можно сконфигурировать HPDZ так, чтобы нефатальные предупреждения не останавливали компиляцию. Если возникли только предупреждения, появится дополнительная кнопка, помеченная CONTINUE (продолжить). Нажатие этой кнопки (или F4) возобновит компиляцию.

Рисунок 3‑6 Окно ошибок

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:56:37 AM.

**┌─ C:\HELLO.C ─────────────────────────────────────────────────────────────────┐**

│#include <stdio.h> **↑**

**│ .**

**│**/\* The infamous hello world program. \*/ **.**

**│ .**

**│**void main(void) **.**

**│**{ **.**

**│** printf(("Hello, world") **.**

**│ .**

**│**}**\_ .**

**│ .**

**│ .**

**│ ◀**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│Line 8/8 Col 2 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ 1 error, 0 warnings from P1.EXE on module C:\HELLO ─────────────────────────┐**

**│; expexted ↑**

**│ ◀**

**│ .**

**│ .**

**│< PREV F9 > < NEXT F10 > < CLEAR F8 > < HELP F1 > < FIX F6 > ↓**

**└←▲ →┘**

В этом случае, сообщение об ошибке ;expected будет выделено, и курсор будет установлен на начало строки после оператора printf(). Это место где была обнаружена первая ошибка. Окно ошибок содержит несколько кнопок, позволяющих выбирать, какую ошибку вы хотите обработать, очистить отображение состояния ошибок или получить объяснение о выделенной в данный момент ошибке. Чтобы получить, сообщение с объяснением ошибки, нажмите щелчком мышью кнопку HELP или нажмите **F1**.

Объяснение ошибки для отсутствующей точки с запятой не предоставляет больше информации, чем мы уже имеем. Тем не менее, объяснения некоторых наиболее необычных ошибок, обнаруживаемых компилятором, могут быть очень полезными. Обрабатываются все ошибки, обнаруженные препроцессором (CPP), проходом 1 (P1), генератором кода (CGxx), ассемблером (ASxx) и компоновщиком (LINK). Вы можете отменить объяснение ошибки, выбрав кнопку HIDE (нажав **Escape** или используя мышь).

В данном случае HPDZ проанализировал ошибку и готов исправить ее. Это обозначено присутствием кнопки *FIX* в нижнем правом углу окна ошибок. Если HPDZ не в состоянии проанализировать ошибку, он не покажет кнопку FIX. Щелчок по кнопке FIX или нажатие **F6** исправит ошибку, добавляя точку с запятой в конец предыдущей строки. Будет воспроизведен звук "бип-бип", и если в окне ошибок присутствует более чем одна строка с ошибкой, HPDZ переместится к следующую ошибке.

Чтобы исправить ошибку вручную, переместите курсор в конец оператора printf() и добавьте отсутствующую точку с запятой. Если у вас есть мышь, просто нажмите левую кнопку в позиции, на которую вы хотите переместить курсор. Если вы используете клавиатуру, переместите курсор с помощью клавиш со стрелками. После того как отсутствующая точка с запятой была добавлена, вы готовы повторно выполнить компиляцию.

На этот раз мы "сократим цикл" редактирование-сохранение-компиляция нажатием F3, чтобы вызвать пункт меню "**Compile and link**". HPDZ автоматически сохранит измененный файл во временный файл, затем скомпилирует его. В окне сообщений будут поочередно отображаться команды, выдаваемые на каждом этапе компиляции. Если все пойдет хорошо, вы услышите тон и увидите сообщение **Compilation successful** (Успешная компиляция).

В этом учебном пособии представлен простой обзор редактирования/компиляции одного разрабатываемого файла. HPDZ также может поддерживать проекты, состоящие из нескольких файлов (включая смешанные источники на языке C и ассемблере), используя возможности проекта. В остальной части этой главы дается подробное описание системы меню, редактора и возможности проекта HPDZ.

## Редактор HPDZ

HPDZ имеет встроенный текстовый редактор для создания и изменения текста программы. Редактор основан на *WordStar* с несколькими незначительными различиями, и некоторыми улучшениями для работы с мышью. Если вы знакомы с WordStar или каким-либо подобным редактором, вы сможете использовать редактор HPDZ без дополнительных инструкций. HPDZ также поддерживает стандартные клавиши PC, и таким образом должен быть легким в использовании знакомым с любыми типичными редакторами MS-DOS или Microsoft Windows.

Редактор HPDZ использует свое собственное окно, известном как *окно редактирования*. Окно редактирования разбито в три области, *рамку*, *область содержимого* и *строку состояния*.

### Рамка

*Рамка* указывает границу между окном редактирования и другими окнами на рабочем столе. Имя текущего файла редактирования отображается в верхнем левом углу рамки. Если редактируется вновь созданный файл, ему будет присвоено имя "untitled" (неназванный). Рамкой можно манипулировать с помощью мыши, позволяя окну перемещаться по рабочему столу и изменять его размер.

### Область содержимого

*Область содержимого* составляет самую большую часть окна и включает редактируемый текст. Когда окно редактирования активно, область содержимого включает курсор, указывающий текущее место вставки. Текстом в области содержимого можно управлять, используя одни только команды клавиатуры, или комбинацию команд клавиатуры и действий мыши. Мышь может использоваться для расположения курсора, прокрутки текста и выбора блоков для операций с буфером обмена.

### Строка состояния

Нижняя строка окна редактирования является *строкой состояния*. Она содержит следующую информацию о редактируемом файле:

* Line показывает текущий номер строки, считая от начала файла, и общее количество строк в файле.
* Col показывает номер столбца, в котором находится курсор, считая от левого края окна.
* Если строка состояния включает текст **^K** после записи Col, это указывает, что редактор ожидает второй символ команды WordStar **ctrl-K**. Смотрите раздел [Команды клавиатуры](#_Клавиатурные_команды) на странице 51 для уточнения списка допустимых команд **ctrl-K**.
* Если строка состояния включает текст **^Q** после запись Col, редактор будет ожидать второго символа команды WordStar **ctrl-Q**. Смотрите раздел [Команды клавиатуры](#_Клавиатурные_команды_1) на странице 51 для уточнения списка допустимых команд **ctrl-Q**.
* Insert указывает, будет ли текст, введенный на клавиатуре, вставлен в позиции курсора. Используя команду *переключателя режима вставки* (клавиша **Ins** на клавиатуре или **ctrl-V**), режим может быть переключен между *Insert* (Вставка) и *Overwrite* (Замещение). В режиме замещения текст вводимый с клавиатуры, перезапишет символы под курсором, вместо того, чтобы вставить их перед курсором.
* Indent указывает, что редактор находится в режиме автоматического отступа. Режим автоматического отступа переключается с помощью нажатия последовательности клавиш **ctrl-Q I**. По умолчанию режим автоматического отступа включен. Когда режим автоматического отступа включен, каждый раз, когда вы добавляете новую строку, курсор выравнивается под первый непустой символ в предыдущей строке. Если редактируемый файл является файлом C, редактор по умолчанию будет в *режиме C*. В этом режиме, когда введена открывающая фигурная скобка ('{'), следующая строка будет расположена с отступом на одну позиция табуляции. Кроме того, он автоматически выровняет закрывающую фигурную скобку ('}') с первым непустым символом в строке, содержащем открывающую скобку. Это делает режим автоматического отступа идеальным для ввода кода C.
* Кнопка **SEARCH** может использоваться для инициирования операции поиска в редакторе. Для выбора SEARCH, щелкните левой кнопкой мыши в любом месте по тексту кнопки. Также поиск может быть активирован, используя клавишу **F2** и последовательность WordStar **ctrl-Q F**.
* Кнопка **NEXT** присутствует только в том случае, если уже была поисковая операция. Она осуществляет поиск вперед следующего вхождения искомого текста. Также NEXT может быть выбрана, используя **shift-F2** или **ctrl-L**.
* Кнопка **PREVIOUS** используется, чтобы искать предыдущее вхождение искомого текста. Эта кнопка присутствует только если уже была поисковая операция. Клавиатурными эквивалентами PREVIOUS являются **ctrl-F2** и **ctrl-P**.

### Клавиатурные команды

Редактор принимает несколько команд с клавиатуры, разбитых на следующие категории: *команды перемещения курсора*, *команды вставки/удаления*, *команды поиска*, *операции с блоками и буфером обмена* и *файловые команды*. Каждая из этих категорий содержит ряд логически связанных команд. Некоторые команды перемещения курсора и операции выбора блока могут также выполняться с помощью мыши.

[Таблица 3-8](#t38) на странице 53 содержит перечень доступных клавиатурных команд и сопоставленные им сочетания клавиш. Ряд команд имеет несколько сопоставлений клавиш, некоторые из них также имеют эквивалентный пункт меню.

Команда Zoom, **ctrl-Q Z**, используется для переключения между оконным и полноэкранным режимом. В полноэкранном режиме HPDZ меню доступно по нажатию клавиши **Alt** или при помощи средней кнопки на трехкнопочной мыши.

### Блочные команды

В дополнение к командам перемещения и редактирования, перечисленным в Таблице ["Клавиши редактирования"](#t38), редактор HPDZ поддерживает также блочные операции в стиле WordStar и управляемые мышью операции с буфером обмена вырезать / копировать/вставить.

Буфер обмена реализован как вторичное окно редактора, что позволяет напрямую вводить и редактировать текст в буфере обмена. Блочные операции с буфером обмена, управляемыми мышью могут свободно смешиваться с операциями вырезать / копировать/вставить в стиле WordStar.

Блочные операции основываются на сочетаниях клавиш ctrl-K и ctrl-Q, которые знакомы любому, кто использовал WordStar совместимый редактор.

[Таблица 3-7](#t37) на странице 52 перечисляет доступные WordStar совместимые блочные операции.

Таблица 3‑7 Клавиши операций с блоками

|  |  |
| --- | --- |
| **Команда** | **Сочетание клавиш** |
| Начало блока | Ctrl-K B |
| Конец блока | Ctrl-K K |
| Срыть/показать блок | Ctrl-K H |
| Перейти к началу блока | Ctrl-Q B |
| Перейти в конец блока | Ctrl-Q K |
| Копировать блок | Ctrl-K C |
| Переместить блок | Ctrl-K V |
| Удалить блок | Ctrl-K Y |
| Прочитать блок из файла | Ctrl-K R |
| Записать блок а файл | Ctrl-K W |

Блочные операции ведут себя обычным способом для редакторов типов WordStar с незначительными отличиями. Блоки "в обратном порядке", с концом блока расположенным перед началом блока, поддерживаются и ведут себя точно как обычное выделение блока. Если блок не выбран, может быть выбран однострочный блок вводом начала блока (**ctrl-K B**) или конца блока (**ctrl-K K**). Если блок уже присутствует, вставка любого начало или конца блока имеет эффект изменения границ блока.

**Начало блока ctrl-K B**

Сочетания клавиш **ctrl-K B** выбирает текущую строку в качестве начала блока. Если блок уже присутствует, маркер начала блока будет перемещен к текущей строке. Если блок не помечен, в текущей строке будет выбран однострочный блок.

**Конец блока ctrl-K K**

Сочетания клавиш **ctrl-K K** выбирает текущую строку в качестве конца блока. Если блок уже присутствует, маркер конца блока будет смещен к текущей строке. Если блок отсутствует, в текущей строке будет выбран однострочный блок.

**Перейти к началу блока ctrl-Q B**

Если блок присутствует, сочетания клавиш **ctrl-Q B** перемещает курсор в строку, содержащую маркер начала блока.

Таблица 3‑8 Клавиши редактирования

|  |  |  |
| --- | --- | --- |
| **Команда** | **Клавиши** | **Клавиши WordStar** |
| **Символ влево** | Стрелка влево | Ctrl-S |
| **Символ вправо** | Стрелка вправо | Ctrl-D |
| **Слово влево** | Ctrl-Стрелка влево | Ctrl-A |
| **Слово вправо** | Ctrl-Стрелка вправо | Ctrl-F |
| **Строка вверх** | Стрелка вверх | Ctrl-E |
| **Строка вниз** | Стрелка вниз | Ctrl-X |
| **Страница вверх** | PgUp | Ctrl-R |
| **Страница вниз** | PgDn | Ctrl-C |
| **В начало строки** | Home | Ctrl-Q S |
| **В конец строки** | End | Ctrl-Q D |
| **К верху окна** |  | Ctrl-Q E |
| **К низу окна** |  | Ctrl-Q X |
| **К началу файла** | Ctrl-Home | Ctrl-Q R |
| **В конец файла** | Ctrl-End | Ctrl-Q C |
| **Переключить режим вставки** | Ins | Ctrl-V |
| **Перенести строку в месте курсора** |  | Ctrl-N |
| **Вставить новую строку ниже курсора** |  | Ctrl-O |
| **Удалить символ в позиции курсора** | Del | Ctrl-G |
| **Удалить символ слева от курсора** | Backspace | Ctrl-H |
| **Удалить строку** |  | Ctrl-Y |
| **Удалить до конца строки** |  | Ctrl-Q Y |
| **Поиск** | F2 | Ctrl-Q F |
| **Поиск вперед** | Shift-F2 | Crtl-L |
| **Поиск назад** | Alt-F2 | Ctrl-P |
| **Переключить режим авто отступа** |  | Ctrl-Q I |
| **Раскрыть/восстановить окно** |  | Ctrl-Q Z |
| **Открыть файл** | Alt-O |  |
| **Новый файл** | Alt-N |  |
| **Сохранить файл** | Alt-S |  |
| **Сохранить файл с новым именем** | Alt-A |  |

**Перейти в конец блока ctrl-Q K**

Если блок присутствует, сочетание клавиш **ctrl-Q K** перемещает курсор в строку, содержащую маркер конца блока.

**Переключатель скрытия блока ctrl-K H**

Переключатель скрытия/отображения блока **ctrl-K H** используется для скрытия или отображения выделения текущего блока. Операции вырезания, копирования, перемещения и удаления блоков возможны когда они отображены. Границы скрытых блоков сохраняются при всех операциях редактирования, таким образом, блок может быть выбран, скрыт и повторно отображен после выполнения других операций редактирования. Обратите внимание, что некоторые блочные и операции с буфером обмена изменяют выделение блока, делая невозможным восстановление изображения ранее скрытого блока.

**Копирование блока ctrl-K C**

Команда **ctrl-K C** вставляет копию текущего выделенного блока перед строкой, которая содержит курсор. Копия блока будет также помещена в буфер обмена. Эта операция эквивалентна операции копирования буфера обмена, с последующей операцией вставки буфера обмена.

**Перемещение блока ctrl-K V**

Команда **ctrl-K V** вставляют текущий блок перед строкой, которая содержит курсор, затем удаляет оригинал блока. Т.е. блок перемещается в новую позицию непосредственно предшествующую текущей строке. Копия блока будет также помещена в буфер обмена. Эта операция эквивалентна операции вырезания буфера обмена с последующей операцией вставки буфера обмена.

**Удаление блока ctrl-K Y**

Команда **ctrl-K Y** удаляет текущий блок. Копия блока будет также помещена в буфер обмена. Эта операция может быть отменена, используя команду вставка буфера обмена. Эта операция эквивалентна команде вырезания буфера обмена.

**Чтение блока из файла ctrl-K R**

Команда **ctrl-K R** запросит пользователя имя текстового файла, который должен быть считан и вставлен перед текущей строкой. Вставленный текст будет выбран как текущий блок. Эта операция может быть отменена, удалением текущего блока.

**Запись блока в файл ctrl-K W**

Команда **ctrl-K** **W** запрашивает у пользователя имя текстового файла, в который будет записан текущий выбранный блок. Эта команда не изменяет выделение блока, текст редактора или буфер обмена.

**Indent** (Отступ)

Эта операция доступна с помощью меню "*Edit*" (Правка). Она вставит отступ на одну позицию табуляции, текущий блок или текущую строку, если блок не выбран.

**Outdent** (Выступ)

Противоположная предыдущей операции, то есть она удаляет один символ табуляции из начала каждой строки в выделенном фрагменте или текущей строки, если блок не выбран. Она доступна только через меню "*Edit*" (Правка).

**Comment/Uncomment** (Комментировать/Раскомментировать)

Также доступная в меню "*Edit*" (Правка), эта операция вставляет или удаляет заголовок комментария стиля C++ (//) в начале каждой строки текущего блока или текущей строки, если блок не выбран. Если строка в настоящее время раскомментирована, она будет закомментирована, и если она уже закомментирована, она будет раскомментирована. Это повторяется для каждой строки в выделенной области. Это позволяет быстро комментировать часть кода во время отладки или тестирования.

### Редактирование буфера обмена

Редактор HPDZ также поддерживает операции с буфером обмена, управляемые мышью, аналогичные тем, которые поддерживаются несколькими хорошо известными графическими пользовательскими интерфейсами.

Текст может быть выбран щелчком мыши и перетаскиванием, удалением, вырезанием или копированием в буфер обмена и вставкой из буфера обмена. Буфер обмена основан на стандартном окне редактора и может непосредственно управляться пользователем. Операции с буфером обмена могут свободно смешиваться с блочными операциями WordStar.

#### Выбор текста

Блоки текста могут быть выбраны с помощью левой кнопки мыши и операций перетаскивания. Могут использоваться следующие операции мыши:

* Один щелчок левой кнопки мыши позиционирует курсор и скроет текущий выбор. Элемент меню "**Hide**" (Скрыть) в меню "**Edit**" (Правка) или команда **ctrl-K H** может использоваться для повторного отображения выбора блока, который был отменен щелчком мыши.
* Двойной щелчок левой кнопки мыши расположит курсор и выберет строку как однострочный блок. Любой предыдущий выбор будет отменен.
* Если левая кнопка нажата и удерживается, можно сделать выбор нескольких строк от позиции щелчка мышью, путем перетаскивания мыши в направлении желаемого выбора. Если мышь переместится вне верхней или нижней границы окна редактора, редактор выполнит прокрутку, позволяя выбрать блок более одной страницы. При отпускании кнопки курсор переместится в позицию мыши. Любой предыдущий выбор будет отменен.

#### Команды буфера обмена

Редактор HPDZ поддерживает несколько команд манипулирования буфером обмена, которые могут использоваться для вырезания текста в буфер обмена, копирования текста в буфер обмена, вставки текста из буфера обмена, удаления текущего выделения и скрытия или отображения текущего выделения. Окно буфера обмена может отображаться и использоваться в качестве вторичного окна редактирования. У многих операций буфера обмена имеются как пункты меню, так и горячие клавиши. Доступны следующие операции с буфером обмена:

**Cut** (Вырезать) **alt-X**

Опция **Cut** копирует текущий выбор в буфер обмена и затем удаляет выбор. Эта действие может быть отменено, используя операцию **Paste** (Вставить). Предыдущее содержимое буфера обмена теряется.

**Copy** (Копировать) **alt-C**

Опция **Copy** копирует текущий выбор в буфер обмена без изменений или удаления выбора. Предыдущее содержимое буфера обмена теряется.

**Paste** (Вставить) **alt-V**

Опция **Paste** вставляет содержание буфера обмена в редактор перед текущей строкой. Содержимое буфера обмена не изменяется.

**Hide** (Скрыть) **ctrl-K H**

Опция **Hide** переключает текущий выбор между скрытым и выведенным на экран состоянием. Эта опция эквивалентна команде WordStar **ctrl-K H**.

**Show clipboard** (Показать буфер обмена)

Этот пункт меню скрывает или отображает окно редактора буфера обмена. Если окно буфера обмена видимо, оно скрывается. Если окно буфера обмена скрыто, оно будет отображено и выбрано в качестве текущего окна. Окно буфера обмена ведет себя как обычное окно редактора во многих отношениях, за исключением того, что не могут быть использованы операции с блоками. Эта опция не имеет эквивалентного сочетания клавиш.

**Clear clipboard** (Очистить буфер обмена)

Эта опция очищает содержимое буфера обмена и не может быть отменена. Если в буфер обмена находится большой выбранный фрагмент, необходимо использовать этот параметр, чтобы сделать доступной редактору дополнительную память после завершения операций с буфером обмена.

**Delete selection** (Удалить выбор)

Этот пункт меню удаляет текущий выбор, не копируя его в буфер обмена. Операцию **Delete selection**, не следует путать с **Cut**, поскольку она не может быть отменена, и копия удаленного текста не сохраняется. Используйте эту опцию, для удаления блока текста, не изменяя содержание буфера обмена.

## Меню HPDZ

Этот раздел содержит описание каждого пункта меню HPDZ. Описание каждого меню включает в себя распечатку экрана, показывающую внешний вид меню на типичном экране HPDZ.

### Меню «» (Система)

Меню **«»** (Система) присутствует во всех HI-TECH оконных приложениях. Оно содержит удобные утилиты для настройки системы и *аксессуары рабочего стола*, которые мы считаем нужно сделать стандартной частью рабочего стола.

**About** **HPDZ** **...** (о HPDZ ...)

Диалоговое окне About HPDZ отображает информацию о номере версии компилятора и сведения о лицензии.

**Setup** **...** (Настройки ...)

Этот пункт меню выбирает стандартное меню конфигурации встроенного микропрограммного обеспечения мыши и присутствует во всех оконных приложениях HI-TECH. Диалоговое окно "mouse setup" (Настройки мыши) позволяет корректировать горизонтальную и вертикальную чувствительность (Sensitivity) мыши, *баллистический порог* мыши[[13]](#footnote-13) (Ballistic threshold) и скорости автоповтора нажатия кнопок мыши (Click repeat rate).

Этот пункт меню недоступен, если драйвер мыши не установлен. С некоторыми ранними драйверами мыши это диалоговое окно будет функционировать не правильно. К сожалению, не существует способа обнаружения подобного поведения драйвера, потому что даже вызов "информации о версии драйвера мыши" отсутствует в некоторых более древних драйверах!

Это диалоговое окно также отобразит информацию о вашей видеокарте и мониторе, используемой версии DOS и доступной памяти DOS. Смотрите [Рисунок 3-2](#r32) на странице 45.

### Меню File (Файл)

Меню **File** (Файл) содержит команды обработки файла, команду HPDZ Quit и список выбора:

**Open ...** (Открыть …) **alt-O**

Эта команда загружает файл в редактор. Вам будет предложено ввести имя файла, и если будет введен шаблон (например, "\*.C"), будет представлено диалоговое окно выбора файла. Если предыдущий редактируемый файл изменен, но не сохранен, у вас будет возможность сохранить его или прервать команду Open.

**New** (Новый) **alt-N**

Команда **New** очищает редактор и создает новый файл для редактирования с именем по умолчанию "untitled". Если предыдущий редактируемый файл редактирования изменен, но не сохранен, будет представлена возможность сохранить его или прервать команду New.

**Save** (Сохранить) **alt-S**

Эта команда сохраняет текущий редактируемый файл. Если файл "untitled", вам будет предложено ввести новое имя, иначе будет использоваться текущее имя файла (отображаемое в рамке окна редактирования).

**Save as ...** (Сохранить как …) **alt-A**

Эта команда аналогична **Save** (Сохранить), за исключением того, что всегда запрашивается имя нового файла.

**Autosave ...** (Автосохранение …)

Этот пункт вызовет диалоговое окно, позволяющее ввести временной интервал в минутах для автоматического сохранения редактируемого файла. Если значение не равно нулю, то текущий редактируемый файл будет автоматически сохраняться во временный файл с определенными интервалами. Если HPDZ завершится ненормально, например, при сбое питания компьютера, при последующем выполнении HPDZ, он автоматически восстановит сохраненную версию файла.

**Quit** (Выход) **alt-Q**

Команда **Quit** используется для выхода из HPDZ в операционную систему. Если текущий редактируемый файл изменен, но не сохранен, будет предоставлена возможность сохранить его или прервать команду Quit.

**Clear pick list** (Очистить список выбора)

Это команда очищает список недавно открытых файлов, которые появляются ниже этой опции.

**Pick list** (Список выбора) **ctrl-F4**

Список выбора содержит список недавно открывавшихся файлов. Файл может быть загружен из списка выбора, выбрав этот файл. Файл открывавшийся последним, может быть восстановлен с использованием сокращения **ctrl-F4**.

### Меню Edit (Правка)

Меню Edit содержит элементы, касающиеся текстового редактора и буфера обмена. Меню редактирования показано в [Таблице 3-7](#r37) на странице 58.

Рисунок 3‑7 Меню Edit HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:58:58 AM.

**┌─ untitled├───────────────────┤───────────────────────────────────────────────┐**

**│ │.Cut alt-X.│ ↑**

**│ │ Copy alt-C │ ◀**

**│ │ Paste alt-V │ .**

**│ │ Hide │ .**

**│ ├───────────────────┤ .**

**│ │ Delete selection │ .**

**│ ├───────────────────┤ .**

**│ │ Search ... │ .**

**│ │ Replace ... │ .**

**│ ├───────────────────┤ .**

**│ │ Show clipboard │ .**

**│ │ Clear clipboard │ .**

**│ ├───────────────────┤ .**

**│ │ Go to line ... │ .**

**│ │ Set tab size ... │ .**

**│ ├───────────────────┤ .**

**│ │ Indent │ .**

**│Line 1│ Outdent │ert C mode < Search F2 > ↓**

**└←▲........├───────────────────┤░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project:│ Comment/uncomment │180 V7.60 ─────────────────────────────────────┐**

**│** HI-**├───────────────────┤**EVELOPMENT SYSTEM (Z80/180/64180 V7.60) **│**

**│** Co**│ C colour coding │** HI-TECH Software, All Rights Reserved **│**

**│ └───────────────────┘** CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

**Cut** (Вырезать) **alt-X**

Опция **Cut** копирует текущий выбор в буфер обмена и затем удаляет выбор. Эта действие может быть отменено, используя операцию **Paste** (Вставить). Предыдущее содержимое буфера обмена теряется.

**Copy** (Копировать) **alt-C**

Опция **Copy** копирует текущий выбор в буфер обмена без изменений или удаления выбора. Предыдущее содержимое буфера обмена теряется.

**Paste** (Вставить) **alt-V**

Опция **Paste** вставляет содержание буфера обмена в редактор перед текущей строкой. Содержимое буфера обмена не изменяется.

**Hide** (Скрыть)

Опция **Hide** переключает текущий выбор между скрытым и выведенным на экран состоянием. Эта опция эквивалентна команде WordStar **ctrl-K H**.

**Delete selection** (Удалить выбор)

Этот пункт меню удаляет текущий выбор, не копируя его в буфер обмена. Операцию **Delete selection**, не следует путать с **Cut**, поскольку она не может быть отменена, и копия удаленного текста не сохраняется. Используйте эту опцию, для удаления блока текста, не изменяя содержание буфера обмена.

**Search ...** (Найти …)

Эта опция вызывает диалоговое окно, позволяющее вводить строку для поиска. Можно выбрать поиск вперед (forwards) или назад (backwards), нажав соответствующую кнопку. Вы можете также решить, должен ли поиск быть чувствительным к регистру (case sensitive) и необходимо ли производить замену замещающей строкой. Эти выборы осуществляется щелком по соответствующим скобкам.

**Replace ...** (Заменить …)

Эта опция - почти такая же как опция **Search**. Она используется, когда вы уверены, что хотите выполнить поиск и замену одной операцией. Можно выбирать между двумя вариантами. Можно искать и затем решать, производить ли замену каждый раз, когда строка поиска найдена. Альтернативно, можно осуществить поиск и замену глобально. Если выбрана опция глобально (global), необходимо быть осторожным в определении строки поиска, поскольку замена не может быть отменена.

**Show clipboard** (Показать буфер обмена)

Этот пункт меню скрывает или отображает окно редактора буфера обмена. Если окно буфера обмена видимо, оно скрывается. Если окно буфера обмена скрыто, оно будет отображено и выбрано в качестве текущего окна. Окно буфера обмена ведет себя как обычное окно редактора во многих отношениях, за исключением того, что не могут быть использованы операции с блоками. Эта опция не имеет эквивалентного сочетания клавиш.

**Clear clipboard** (Очистить буфер обмена)

Эта опция очищает содержимое буфера обмена и не может быть отменена. Если в буфер обмена находится большой выбранный фрагмент, необходимо использовать этот параметр, чтобы сделать доступной редактору дополнительную память после завершения операций с буфером обмена.

**Go to line ...** (Перейти к строке …)

Команда **Go to line** позволяет перейти непосредственно к любой строке в текущем редактируемом файле. Появится диалоговое окно с приглашением ввести номер строки. В названии диалогового окна указывается допустимый диапазон номеров строк в исходном файле.

**Set tab size ...** (Установить размер табуляции …)

Эта команда используется, чтобы установить размер позиций табуляции в редакторе. Размер табуляции по умолчанию равняется 8, можно использовать значения от 1 до 16. Для нормального исходного кода C значение 4 является подходящим выбором. Размер табуляции будет сохранен как часть вашего проекта, если вы будете использовать средство Make.

**Indent** (Отступ)

Эта операция доступна с помощью меню "*Edit*" (Правка). Она вставит отступ на одну позицию табуляции, текущий блок или текущую строку, если блок не выбран.

**Outdent** (Выступ)

Противоположная предыдущей операции, то есть она удаляет один символ табуляции из начала каждой строки в выделенном фрагменте или текущей строки, если блок не выбран.

**Comment/uncomment** (Комментировать / Раскомментировать)

Этот пункт вставляет или удаляет начальные символы комментария в стиле C++ (//) в каждой строки текущего блока или текущей строки. Это приводит к комментированию этих строк кода, предотвращая их компиляцию. Если строка закомментирована этим способом, начальные символы комментария будут удалены.

**C colour coding** (Подсветка синтаксиса C)

Эта опция переключает цветное кодирование текста в окне редактора. Она включает и выключает цвета для различных типов текста. Перед этим пунктом появляется метка, когда она активна. Для полного описания цветов, используемых в HPDZ и как выбрать определенные схемы, вы должны обратиться к [Разделу 3.2.2](#_Цвета) "*Цвета*" на странице 38.

### Меню Options (Опции)

Меню Options (опции) содержит команды, которые позволяют выбрать опции компилятора, модель памяти, и целевой процессор. Выборы, сделанные в этом меню, будут сохранены в используемом файле проекта. Меню Options показано на [Рисунке 3-8](#r38) на странице 60.

**Memory model and chip type ...** (Модель памяти и тип микросхемы …)

Эта опция активирует диалоговое окно, позволяющее выбрать модель памяти и тип используемого процессора.

Доступны малая и с переключением банков (большая) модели памяти. Для получения дополнительной информации о моделях памяти смотрите [Раздел 4.2.10](#_-Bs:_Выбор_малой) "*-Bs: Выбор малой модели памяти*" на странице 79 и [Раздел 4.2.11](#_-Bl:_Выбор_большой) "*-Bl: Выбор большой модели памяти*" на странице 79.

Рисунок 3‑8 Меню Options HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:57:18 AM.

**┌─ untitled ─────├────────────────────────────────┤────────────────────────────┐**

**│ │ Memory model and chip type ... │ ↑**

**│ ├────────────────────────────────┤ ◀**

**│ │ CP/M-80 output file ... │ .**

**│ │♦ROM output file ... │ .**

**│ │ ROM & RAM addresses ... │ .**

**│ │ ROM chechsum specs ... │ .**

**│ ├────────────────────────────────┤ .**

**│ │ Long formats in printf() │ .**

**│ │ Float formats in printf() │ .**

**│ │ Map and symbol file options ...│ .**

**│ ├────────────────────────────────┤ .**

**│ │♦Save dependency information │ .**

**│ └────────────────────────────────┘ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│Line 1/1 Col 1 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180 V7.60 ─────────────────────────────────────┐**

**│** HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/180/64180 V7.60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

Доступны процессоры Z80 и Z180/64180.

Это диалоговое окно также позволяет выбирать, будет ли использоваться альтернативный набор регистров в обычных функциях. Для того, чтобы установка возымела действие, должна также использоваться глобальная оптимизация. При действии этой установки, компилятор будет использовать альтернативный набор регистров для хранения переменных в случае необходимости. Для получения дополнительной информации смотрите [Раздел 4.2.5](#_-ALTREG:_Использовать_альтернативный) "*-ALTREG: Использовать альтернативный набор регистров*" на странице 78.

Вы можете также выбрать, хотите ли использовать 8-разрядную адресацию портов ввода-вывода, вместо значения по умолчанию, 16-разрядной адресации портов? Для получения дополнительной информации смотрите [Раздел 4.2.34](#_-P8:_Использовать_8) "*-P8: Использовать 8 разрядную адресацию портов*" на странице 86.

**CP/M-80 output file ...** (Выходной файл CP/M-80 …)

Вы можете выбрать вывод в .COM файл CP/M-80 или файл библиотеки.

**ROM output file ...** (Выходной файл ПЗУ …)

Тип выходного файла по умолчанию - Intel HEX. Другими выборами являются: HEX S-записи Motorola, двоичный образ, UBROF, Tektronix HEX, American Automation symbolic HEX, Intel OMF-51 и Bytecraft .COD. Эта опция также позволит указать, что необходимо создать библиотеку, а не .EXE файл. Библиотека может быть создана только из файла проекта.

**ROM & RAM addresses ...** (Адреса ПЗУ и ОЗУ …)

Эта опция позволяет вводить адреса ПЗУ и ОЗУ в целевой системе. Для уточнения стандартных значений, смотрите инструкции в [Разделе 3.3](#_Учебник:_создание_и) "*Учебник*" на странице 45.

Информация, введенная в это диалоговое окно, используется компоновщиком для назначения адресов вашему коду. Обратите внимание, что некоторые из адресов ОЗУ могут быть обнулены, чтобы позволить HPDZ устанавливать их автоматически. Адреса используются следующим образом:

*ROM & RAM addresses* (Адреса ПЗУ и ОЗУ)

Адреса ПЗУ и ОЗУ применимы и к маленькой и к большой (с переключением банков) моделям памяти.

*ROM address* (Адрес ПЗУ): Это адрес в ПЗУ, где должен начаться код программы. Практически для всех систем Z80 он будет адресом 0, наименьшим доступным адресом ПЗУ. При компиляции кода, который должен быть загружен в ОЗУ, используя отладчик LUCIFER, этот адрес будет началом загружаемой области ОЗУ.

Область ПЗУ начинается с вектора сброса, за которым следуют любые другие векторы прерываний, которые были инициализированы с помощью макросов обработки векторов прерываний, определенных в <intrpt.h>.

*RAM address* (Адрес ОЗУ): Это начальный адрес ОЗУ. Для систем Z180 он должен быть адресом в пределах первых 64 Кбайт, в которые ОЗУ должна быть отображена (т.е. *логический* адрес ОЗУ).

*RAM size* (Размер ОЗУ): Это размер в байтах доступной оперативной памяти, начинающийся с *Адреса ОЗУ*.

*NVRAM address* (Адрес NVRAM): Это адрес блока энергонезависимого ОЗУ, которое будет использоваться для постоянных переменных. Если все ОЗУ энергонезависимое, или постоянные переменные не используются, это значение должно быть нулем. Это приведет к размещению энергонезависимой области ОЗУ в стандартной области ОЗУ. Значение по умолчанию равно нулю.

Для системы Z80, использующей маленькую модель, требуются только эти четыре значения (*ROM address*, *RAM address*, *RAM size* и *NVRAM address*).

**Physical addresses** (Физические адреса)

Физические адреса применимы только для большой (с переключением банков) модели памяти.

*RAM address* (Адрес ОЗУ): физический адрес (20-разрядный), по которому ОЗУ фактически находится в системе Z180. Чтобы использовать ОЗУ, оно должно быть отображено в нижние 64 Кбайт памяти. Это значение требуется для кода большой модели и для кода Z180, если ОЗУ адресуется за пределы нижних 64 Кбайт.

*Banked area address* (Адрес переключаемой области): Начальный адрес ПЗУ, который должен быть отображен в переключаемую область.

**Banked area logical addresses** (Логические адреса переключаемой области)

Логические адреса переключаемой области применимы только для большой (с переключением банков) модели памяти.

*Banked area address* (Адрес переключаемой области): логический начальный адрес переключаемой области, в нижем 64 Кб адресном пространстве. Это и следующее значение, banksize, определяют окно в нижней части 64 Кбайт, которое отображается в ПЗУ по различным физическим адресам.

*Banked area size* (Адрес переключаемой области): это размер переключаемой области, которая отображается в ПЗУ по различным физическим адресам.

В этом диалоговом окне вы также можете выбрать, копируются ли инициализированные данные из ПЗУ в ОЗУ (по умолчанию) или остаются в ПЗУ.

**Long formats in printf()** (Форматы long в printf)

Этот опция используется, чтобы сообщить компоновщику, что вы хотите использовать библиотеку с функцией printf() поддерживающей вывод значений long. Библиотека long включает версию printf(), поддерживающую форматы вывода значений long %1d, %1u и %1x. При использовании этой опции, скомпилированное приложение увеличится в размере. Эту опцию следует использовать только если необходимо использовать форматы long в printf(), она не требуется, если вы просто хотите выполнить вычисления длинного целого. При выборе этой опции в строке меню появится маркер.

**Float formats in printf()** (Форматы float в printf)

Этот опция используется, чтобы сообщить компоновщику, что вы хотите использовать библиотеку с функцией printf(), поддерживающей вывод значений с плавающей точкой. Библиотека float включает версию printf(), поддерживающую форматы вывода значений с плавающей точкой %e, %f и %g. При использовании этой опции, скомпилированное приложение увеличится в размере. Она не требуется для выполнения вычислений с плавающей точкой, поэтому используйте ее, только если необходимо использовать форматы с плавающей точкой в printf().

**Map and Symbol File Options ...**

Это диалоговое окно позволяет устанавливать различные опции относящиеся к отладочной информации, файлу распределения памяти (map) и файлу символов.

**Source level debug info**

Этот пункт меню используется, чтобы включить или отключить отладочную информацию исходного уровня в текущем файле символов. Если вы используете отладчик программного обеспечения HI-TECH Software LUCIFER в качестве внутрисхемного эмулятора, следует включить эту опцию.

**Sort map by address** (Сортировка распределения памяти по адресу)

По умолчанию таблица символов в распределении ссылок сортироваться по имени. Эта опция предписывает сортировать его численно, в зависимости от значения символов.

**Suppress local symbols** (Исключить локальные символы)

Предотвращает включение всех локальных символов в файл символов. Даже если эта опция не будет активна, компоновщик будет фильтровать сгенерированные компилятором несущественные символы из файла символов.

**Avocet format symbol file** (Файл символов в формате Avocet)

Используйте эту опцию для генерации avocet AVSIM совместимого файл символов.

### Меню Compile (Компилировать)

Меню Compile, показанное на [Рисунке 3-9](#r39) на странице 64, содержит различные формы команд компиляции вместе с несколькими независимыми от машины опциями настроек компилятора.

**Compile and link** (Скомпилировать и скомпоновать) **F3**

Эта команда скомпилирует один исходный файл и затем вызовет компоновщик и objtohex.exe для создания исполняемого файла. Если исходный файл является файлом .AS, он будет передан непосредственно ассемблеру. Выходной файл будет иметь то же базовое имя, что и исходный файл, но другое расширение. Например, HELLO.C будет скомпилирован в HELLO.EXE

**Compile to .OBJ** (Скомпилировать в объектный файл) **shift-F3**

Компилирует один исходный файл только в файл .OBJ. Компоновщик и objtohex не вызываются. Файлы .AS передаются непосредственно ассемблеру. Полученный объектный файл будет иметь то же базовое имя, что и исходный файл и расширение .OBJ.

**Compile to .AS** (Скомпилировать в ассемблерный файл) **shift-F3**

Этот пункт меню компилирует один исходный файл на язык ассемблера, создавая файл ассемблера с тем же базовым именем, что и исходный файл, и расширение .AS. Эта опция удобна, если необходимо исследовать или изменить код, сгенерированный компилятором. Если текущий исходный файл является файлом .AS, ничего не произойдет.

**Stop on Warnings** (Остановка на предупреждении)

Этот переключатель определяет, будет ли остановлена компиляция при обнаружении нефатальных ошибок. При выборе этого пункта перед ним появляется маркер.

**Warning level ...** (Уровень предупреждений) **alt-W**

Эта команда вызывает диалоговое окно, которое позволяет определить уровень предупреждений компилятора, т.е. она определяет, насколько придирчив компилятор к допустимому, но сомнительному коду. В настоящее время реализован диапазон уровней предупреждений от -9 до 9, где более низкие уровни предупреждений являются более строгими. На уровне 9 подавляются все предупреждения (но не ошибки). Уровень 1 подавляет сообщения "Func() declared implicit int" (функция неявно объявлена int), которое распространено, при компиляции кода предназначенного для Unix. Уровень 3 предназначен для компиляции кода, написанного для менее строгих (и K&R) компиляторов. Уровень 0 является значением по умолчанию. Эта команда эквивалентна опции -W команды ZC.

Рисунок 3‑9 Меню Compile HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:57:18 AM.

**┌─ untitled ──────────────├─────────────────────────────┤──────────────────────┐**

**│ │ Compile and link F3 │ ↑**

**│ │ Compile to OBJ shift-F3 │ ◀**

**│ │ Compile to AS ctrl-F3 │ .**

**│ ├─────────────────────────────┤ .**

**│ │♦Stop on warnings │ .**

**│ │ Warning level ... alt-W │ .**

**│ │ Optimization ... alt-Z │ .**

**│ │ Identifier length ... │ .**

**│ ├─────────────────────────────┤ .**

**│ │ Pre-process assembler files │ .**

**│ │♦Generate assembler listings │ .**

**│ │ Generate C source listings │ .**

**│ └─────────────────────────────┘ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│Line 1/1 Col 1 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180 V7.60 ─────────────────────────────────────┐**

**│** HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/180/64180 V7.60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

**Optimization ...** (Оптимизация …) **alt-Z**

Выбор этого элемента откроет диалоговое окно, позволяющее выбрать различные виды и уровни оптимизации. По умолчанию оптимизация не выполняется. Выбранные в этом диалоговом окне параметры будут сохранены в файле проекта, если он используется.

**Identifier length ...** (Длина идентификатора …)

По умолчанию идентификаторы C считаются значимыми только до 31 символа. Эта команда позволяет установить число используемых значимых символов между 31 и 255.

**Pre-process assembler files** (Ассемблерные файлы препроцессора)

При выборе этого пункта HPDZ будет передавать ассемблерные файлы через препроцессор перед ассемблированием. Это позволяет использовать макросы препроцессора C и условные операторы в ассемблерных файлах. При выборе этого пункта перед ним появляется маркер.

**Generate assembler listing** (Генерация листинга препроцессора)

Этот пункт меню предписывает ассемблеру генерировать файл листинга для каждого компилируемого C или ассемблерного исходного файла. Имя файла листинга определяется именем файла символов, например TEST.C произведет файл листинга названный TEST.LST.

**Generate C source listings** (Генерация исходного листинга C)

Выбор этой опции приведет к созданию исходного листинга C для каждого скомпилированного файла C. Файл листинга будет называться точно так же как и файл ассемблера (описанный выше), но будет содержать исходный код C с номерами строк и расширенными символами табуляции. Настройки расширения табуляции берутся из настроек позиций табуляции редактора.

### Меню Make (Создать)

Меню **Make** ([Рисунок 3-10](#r310) на странице 65) содержит все команды, необходимые для использования средства проект (project) HPDZ. Средство проект позволяет легко создавать сложные приложения с несколькими исходными файлами, а также обеспечивает высокую степень контроля над некоторыми встроенными функциями и утилитами компилятора. Чтобы использовать проект, необходимо выполнить несколько шагов.

* Создайте новый файл проекта, используя команду **New project ...** (Новый проект...). После выбора имени файла проекта HPDZ представит несколько диалоговых окон, которые позволят вам настроить модель памяти.
* Введите список имен исходных файлов, используя команду **Source file list ...** (Список исходных файлов...).
* Установите любые специальные библиотеки, предопределенные символы препроцессора, объектные файлы или опции компоновщика, используя другие пункты меню **Make**.
* Сохраните файл проекта, используя команду **Save project** (Сохранить проект).
* Скомпилируйте проект, используя команду **Make** (Создать) или **Re-make** (Пересоздать).

Рисунок 3‑10 Меню Make HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:57:18 AM.

**┌─ untitled ───────────────────────├────────────────────────────┤──────────────┐**

**│ │ Make F5 │ ↑**

**│ │ Re-make ctrl-F5 │ ◀**

**│ │ Re-link shift-F5 │ .**

**│ ├────────────────────────────┤ .**

**│ │ Load project ... alt-P │ .**

**│ │ New project ... │ .**

**│ │ Save project │ .**

**│ │ Rename project ... │ .**

**│ ├────────────────────────────┤ .**

**│ │ Output file name ... │ .**

**│ │ Map file name ... │ .**

**│ │ Symbol file name ... │ .**

**│ ├────────────────────────────┤ .**

**│ │ Source file list ... │ .**

**│ │ Object file list ... │ .**

**│ │ Library file list ... │ .**

**│ │ CPP pre-defined symbols ...│ .**

**│Line 1/1 Col 1 Insert│ CPP include paths ... │ ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░│ Linker options ... │░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180│ Objtohex options ... │──────────────┐**

**│** HI-TECH C PROFESSIONAL DEVE**└────────────────────────────┘**V7.60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

**Make** (Создать) **F5**

Команда **Make** повторно компилирует текущий проект. При выборе **Make** HPDZ повторно компилирует все исходные файлы, которые были изменены со времени последнего запуска команды **Make**. HPDZ определяет, необходимо ли перекомпилировать исходный файл, проверяя дату и время изменения исходного файла и соответствующего объектного файла. Если дата и время последнего изменения исходного файла более поздние, чем у объектного файла, он будет заново скомпилирован.

Если все файлы .OBJ текущие, но выходной файл не найден, HPDZ повторно скомпонует, имеющиеся объектные файлы. Если все объектные файлы являются текущими, и выходной файл присутствует и актуален, HPDZ распечатает сообщение в окне сообщения, указывающее, что ничто не сделано.

HPDZ также автоматически проверит зависимости, т.е. он отсканирует исходные файлы, чтобы определить, какие файлы включены и будет включать эти файлы в тест, чтобы определить, должен ли файл быть перекомпилирован. Другими словами, если изменен заголовочный файл, любые исходные файлы включающие этот заголовочный файл будут заново скомпилированы.

Если вы забудете использовать список исходных файлов для выбора включаемых файлов, HPDZ отобразит диалоговое окно с предупреждением, что файлы не были выбраны. Затем вам нужно будет выбрать кнопку **Done** (Готово) или нажать клавишу **Escape** (Выйти). Это вернет вас в окно редактора.

**Re-make** (Создать повторно) **ctrl-F5**

Команда **Re-make** принудительно перекомпилирует все исходные файлы текущего проекта. Эта команда эквивалентна удалению всех файлов .OBJ и последующему выбору **Make**.

**Load project ...** (Загрузить проект …) **alt-P**

Эта команда загружает предопределенный файл проекта. Перед вами появится диалоговое окно выбора файла, позволяющее выбрать и загрузить файл .PRJ. Если эта команда выбрана, когда текущий проект был изменен, но не сохранен, будет предоставлена возможность сохранить проект или прервать загрузку проекта. После загрузки файла проекта заголовок окна сообщения будет изменен, чтобы отобразить имя файла проекта.

**New project ...** (Новый проект …)

Эта команда позволяет пользователю начать новый проект. Вся текущая информация о проекте очищается, и все элементы в меню **Make** включены. Пользователю предоставляется возможность сохранить любой текущий проект и затем предложено ввести имя нового проекта.

После ввода нового имени HPDZ представит несколько диалоговых окон, позволяющих настроить проект. Эти диалоговые окна позволят выбирать: тип процессора, модель памяти и тип с плавающей точкой; тип выходного файла; Адреса ПЗУ и ОЗУ, параметры настройки оптимизации и опции файлов распределения памяти (map) и символов. Вам все равно придется вводить имена исходных файлов в списке исходных файлов.

**Save project** (Сохранить проект)

Этот пункт сохраняет текущий проект в файл.

**Rename project ...** (Переименовать проект …)

Это позволит определять новое имя проекта. При последующих сохранениях проект будет сохранен с новым именем файла. Существующий файл проекта не будет затронут, если он был уже сохранен.

**Output file name ...** (Имя выходного файла …)

Эта команда позволяет пользователю выбрать имя выходного файла компилятора. Это имя автоматически настраивается при создании проекта. Например, если создается проект с названием PROG1 и создается файл .EXE, имени выходного файла будет автоматически присвоено PROG1.EXE.

**Map file name ...** (Имя файла распределения памяти …)

Эта команда позволяет пользователю включать генерацию карты распределения памяти символов текущего проекта и определить имя файла. Наличие символа маркера перед этим пунктом означает, что выбрана генерация файла распределения памяти. По умолчанию имя файла распределения памяти создается из названия проекта, например, PROG1.MAP.

**Symbol file name ...** (Имя файла символов …)

Эта команда позволяет вам выбрать генерацию файла символов и спецификацию имени файла символов. По умолчанию имя файла символов будет создано из имени проекта, например. PROG1.SYM. Выбранный файл символов подходит для использования с любым отладчиком HI-TECH Software.

**Source file list ...** (Список исходных файлов …)

Эта опция отображает диалоговое окно, позволяющее редактировать список исходных файлов. Исходные файлы проекта должны быть введены в список по одному в строке. По окончании ввода из списка исходных файлов можно выйти, нажав Escape, щелкнув мышью по кнопке DONE или щелкнув мышью в строке меню.

Список исходных файлов может содержать любое сочетание исходных файлов языка С и ассемблера. Исходные файлы C должны иметь суффикс .C, а файлы на языке ассемблера - суффикс .AS, с тем что бы HPDZ мог определить, куда нужно передавать файлы.

**Object file list ...** (Список объектных файлов …)

Этот параметр позволяет добавлять любые дополнительные файлы .OBJ в проект. Вводите только один файл .OBJ в строке. Работа этого диалогового окна совпадает с диалоговым окном списка исходных файлов. Этот список обычно содержит только один объектный файл: модуль запуска времени выполнения для текущей модели генерации кода. Например, если проект генерирует код малой модели для Z80, по умолчанию этот список будет содержать модуль запуска (startoff) rtz80-s.obj. Объектные файлы, соответствующие файлам в списке исходных файлов, **не должны** вводиться здесь, поскольку автоматически используются файлы .OBJ, сгенерированные из исходных файлов. Этот список следует использовать только для дополнительных файлов .OBJ, для которых недоступен исходный код, например, код запуска времени выполнения или служебные функции, импортированные из внешнего источника.

Если необходимо связать большое количество файлов .OBJ, их следует уплотнить в один файл .LIB с помощью утилиты LIBR, и затем получить доступ, используя команду Library file list ... (Список библиотечных файлов ...).

**Library file list ...** (Список библиотечных файлов …)

Эта команда позволяет выполнять поиск любых дополнительных библиотек объектных кодов при компоновке проекта. Этот список обычно содержит только библиотеки по умолчанию для используемой модели памяти. Например, если текущий проект генерирует код Z80 для малой модели памяти и printf, поддерживающей вывод значений с плавающей точкой, этот список будет содержать библиотеки z80-sc.lib и z80-sf.lib. Если требуется дополнительная библиотека, добавленная из внешнего источника, она должна быть введена здесь.

Рекомендуется вводить любые нестандартные библиотеки до стандартных библиотек C, если они ссылаются на стандартные библиотечные процедуры. Обычный порядок библиотек должен быть следующим: пользовательские библиотеки, библиотека операций с плавающей точкой, стандартная библиотека C. Библиотека операций с плавающей точкой должна быть соединена перед стандартной библиотекой C, если используются переменные с плавающей точкой. Иногда необходимо несколько раз сканировать пользовательскую библиотеку. В этом случае необходимо ввести имя библиотеки несколько раз.

**CPP pre-defined symbols ...** (Предопределенные символы препроцессора …)

Эта команда позволяет определить любые специальные предопределенные символы. Каждая строка в этом списке эквивалентна опции -D компилятору командной строки ZC. Например, если нужно определить макрос CPP с именем DEBUG со значением 1, добавьте в этот список строку DEBUG=1. Некоторые стандартные символы будут предварительно определены в этом списке, их не следует удалять, поскольку некоторые стандартные файлы заголовков зависят от их присутствия.

**CPP include paths ...** (Пути поиска включаемых файлов …)

Эта опция позволяет препроцессору C осуществлять поиск заголовочных файлов в дополнительных каталогах. Если включаемый заголовочный файл заключен в угловые скобки, например <stdio.h>, компилятор будет осуществлять поиск в каждом каталоге из этого списка, пока не найдет файл.

**Linker options ...** (Опции компоновщика …)

Эта команда позволяет изменять опции, передаваемые компоновщику HPDZ. Содержимое по умолчанию командной строки компоновщика генерируется компилятором из информации, выбранной в меню Options (Опции): модель памяти и т.д. Вы должны использовать эту команду, только если уверены, что знаете, что делаете!

**Objtohex options ...** (Опции objtohex …)

Эта команда позволяет изменять опции, передаваемые от HPDZ к objtohex. Обычно изменять эти опции не требуется, так как генерация двоичных файлов и файлов HEX может быть выбрана в меню Option. Однако, если вы хотите сгенерировать один из необычных форматов вывода, который может произвести objtohex, например COFF, вам нужно изменить опции с помощью этой команды.

### Меню Run (Выполнить)

Меню Run, показанное на [Рисунке 3-11](#r311) на странице 69, содержит опции, позволяющие запускать команды MS-DOS и пользовательские программы. Оно также содержит опции позволяющие выполнять код, используя отладчик LUCIFER.

**DOS command ...** (Команда DOS …) **alt-D**

Эта опция позволяет выполнить команду DOS точно так же, как будто она была введена в приглашении COMMAND.COM. Эта команда может быть внутренней командой DOS, например DIR, или имя программы, которую нужно выполнить. Если необходимо выполнить командный процессор DOS, используйте команду DOS Shell ниже.

**Предупреждение**: не используйте эту опцию для загрузки резидентных программ TSR.

**DOS Shell ...** (Командный процессор DOS …) **alt-J**

Этот пункт вызовет оболочку DOS COMMAND.COM, т.е. вы сразу увидите приглашение командной строки DOS, в отличие от пункта DOS command (команда DOS), которая запрашивает команду. Для возврата в HPDZ, введите "exit" (выход) в командной строке DOS.

Рисунок 3‑11 Меню Options HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 11:59:59 AM.

**┌─ untitled ─────────────────────────────├─────────────────────────────┤───────┐**

**│ │ DOS command ... alt-D │ ↑**

**│ │ DOS Shell ... alt-J │ ◀**

**│ ├─────────────────────────────┤ .**

**│ │ Download ... │ .**

**│ │ Debugger ... │ .**

**│ ├─────────────────────────────┤ .**

**│ │ Optimization ... alt-Z │ .**

**│ │ Identifier length ... │ .**

**│ └─────────────────────────────┘ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│ .**

**│Line 1/1 Col 1 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180 V7.60 ─────────────────────────────────────┐**

**│** HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/180/64180 V7.60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

**Download ...** (Загрузка …)

Эта опция запускает отладчик LUCIFER, автоматически загружает текущий выходной файл и загружает текущий файл символов. Если отладчик не настроен, эта опция недоступна. Вместо этого следует использовать команду Debugger setup ... (Настройка отладчика...). LUCIFER может загрузить фалы только Motorola HEX, Intel HEX и двоичные. Не следует пытаться загружать файлы любых других типов.

**Debugger ...** (Отладчик …)

Эта опция запускает отладчик LUCIFER с текущим файлом символов. Эта опция не загружает код пользователя, поэтому может использоваться для возвращения в приостановленный сеанс LUCIFER.

**Debugger setup ...** (Настройка отладчика …)

Эта опция активирует диалоговое окно, которое позволяет выбирать параметры последовательного порта для отладчика LUCIFER. После задания параметры сохраняются в файле проекта вместе с другими настройками. Настройки по умолчанию для LUCIFER версии Z80 сохраняются в переменной окружения LUCZ80\_ARGS. Для получения дополнительной информации обратитесь к [Главе 8](#_Отладчик_уровня_исходного) "*Отладчик уровня исходного кода Lucifer*" на странице 161.

**Auto download after compile** (Автоматическая загрузка после компиляции)

Эта опция позволяет включать или отключать автоматическую загрузку кода после компиляции. Когда она включена, и отладчик LUCIFER настроен, при успешной компиляции HPDZ автоматически вызывает LUCIFER, чтобы загрузить текущий выходной файл и текущий файл символов. Когда она включена перед этим пунктом появляется маркер.

### Меню Utility (Утилиты)

Меню Utility ([Рисунок 3-12](#r312) на странице 70) содержит некоторые полезные утилиты, включенные в HPDZ.

Рисунок 3‑12 Меню Utility HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 12:00:50 AM.

**┌─ untitled ──────────────────────────────────├────────────────────────────┤───┐**

**│ │ String search ... │ ↑**

**│ ├────────────────────────────┤ ◀**

**│ │ Memory usage map │ .**

**│ ├────────────────────────────┤ .**

**│ │ Calculator │ .**

**│ │ Ascii Table │ .**

**│ ├────────────────────────────┤ .**

**│ │ Identifier length ... │ .**

**│ ├────────────────────────────┤ .**

**│ │ User command 1 shift-F7 │ .**

**│ │ User command 1 shift-F8 │ .**

**│ │ User command 1 shift-F9 │ .**

**│ │ User command 1 shift-F10 │ .**

**│ └────────────────────────────┘ .**

**│ .**

**│ .**

**│ .**

**│Line 1/1 Col 1 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180 V7.60 ─────────────────────────────────────┐**

**│** HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/180/64180 V7.60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

**String search ...** (Поиск строки …)

Эта опция позволяет вам выполнить поиск строки в списке файлов. Опция выдает диалоговое окно, которое позволяет вам ввести искомую строку, а затем выбрать список файлов для поиска. Вы также можете выбрать чувствительность к регистру. Можно ограничить поиск для списка исходных файлов или только текущим проектом.

**Memory usage map** (Карту использования памяти)

Эта опция отображает окно, которое содержит подробную карту использования памяти последней скомпилированной программы.

Окно карты использования памяти можно закрыть, щелкнув мышью по полю закрытия в верхнем левом углу рамки окна, или нажав клавишу **Esc**, пока окно карты памяти является окном переднего плана.

**Calculator** (Калькулятор)

Эта команда выбирает калькулятор программиста HI-TECH Software. Это целочисленный калькулятор с несколькими дисплеями, способный выполнять вычисления в 2 (двоичной), 8 (восьмеричной), 10 (десятичной) и 16 (шестнадцатеричной) системе счисления. Результаты каждого вычисления отображается во всех четырех основаниях одновременно.

Работает точно так же, как "реальный" калькулятор - просто нажимайте кнопки! Если имеется мышь, можно щелкать по кнопкам на экране, или просто использовать клавиатуру. Большие кнопки справа от дисплея позволяют выбирать, какое основание используется для ввода чисел.

Окно калькулятора может быть перемещено по желанию, и таким образом может быть оставлено на экране, во время использования редактора. Окно калькулятора можно закрыть, нажав кнопку OFF в нижнем правом углу, щелкнув по полю закрытия в верхнем левом углу рамки окна, или нажав **Esc**, в момент когда калькулятор является оном переднего плана.

**Ascii Table** (Таблица символов Ascii)

Эта опция выбирает окно, содержащее таблицу символов ASCII. Окно таблицы ASCII содержит четыре кнопки, которые позволяют закрыть окно или выбрать отображение таблицы в восьмеричном, десятичном или шестнадцатеричном формате.

Окно таблицы ASCII можно закрыть, нажав кнопку CLOSE в нижнем левом углу, щелкнув по полю закрытия в верхнем левом углу рамки, или нажав **Esc**, когда окно таблицы ASCII находится на переднем плане.

**Define user commands ...** (Команды определенные пользователем …)

Таблица 3‑9 Макросы, используемые в командах пользователей

| **Имя макроса** | **Назначение** |
| --- | --- |
| **$(LIB)** | Расширяется до имени каталога с файлом системной библиотеки; например, C:\HPDZ\LIB\ |
| **$(CWD)** | Текущий рабочий каталог |
| **$(INC)** | Имя каталога с системными включаемыми файлами |
| **$(EDIT)** | Имя файла загруженного в настоящее время в редактор. Если текущий файл был изменен, оно будет заменено именем автоматически сохраненного временного файла. При возврате оно будет перезагружено, если оно изменилось. |
| **$(OUTFILE)** | Имя текущего выходного файла , т.е. исполняемого файла. |
| **$(PROJ)** | Базовое имя текущего проекта, например, если текущий файл проекта - AUDIO.PRJ, этот макрос будет расширяться до AUDIO без точки или типа файла. |

Меню Utility содержит четыре определяемых пользователем команды. Этот пункт вызовет диалоговое окно, позволяющее определять эти команды. По умолчанию команды выделены серым цветом (недоступны), но будут включены при определении команды. Каждая команда имеет форму команды DOS с доступными макроподстановками. Доступные макросы перечислены в [Таблице 3-9](#t39) на странице 71. Каждой определяемой пользователем команде сопоставлена горячая клавиша. Это shift-F7 – shift-F10 для команд 1 - 4. При выполнении команды пользователя, текущий редактируемый файл, если изменен, будет сохранен во временный файл и макрос $(EDIT) будет отражать имя сохраненного временного файла, а не оригинальное имя. При возврате, если временный файл был изменен, он будет перезагружен в редактор. Это позволяет легко интегрировать внешний редактор в HPDZ.

### Меню Help (Справка)

Меню Help ([Рисунок 3-13](#r313) на странице 72) содержит пункты, позволяющие получить помощь по всем перечисленным темам.

Рисунок 3‑13 Меню Help HPDZ

..**«»** **F**ile **E**dit Op**t**ions Comp**i**le **M**ake **R**un **U**tility **H**elp 12:01:03 PM.

**┌─ untitled ───────────────────────────────────────├──────────────────────────┤┐**

**│ │ HPDZ │↑**

**│ │ C Library Reference │◀**

**│ ├──────────────────────────┤.**

**│ │ Editor Keys │.**

**│ ├──────────────────────────┤.**

**│ │ Technical Support │.**

**│ ├──────────────────────────┤.**

**│ │ ZC Compiler Options │.**

**│ │ Z80/Z180 Instruction Set │.**

**│ │ Z180/64180 I/O Registers │.**

**│ ├──────────────────────────┤.**

**│ │ Release notes │.**

**│ └──────────────────────────┘.**

**│ .**

**│ .**

**│ .**

**│ .**

**│Line 1/1 Col 1 Insert C mode < Search F2 > ↓**

**└←▲...........░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░→┘**

**┌─ Project: WOW.PRJ (Z80/Z180/64180 V7.60 ─────────────────────────────────────┐**

**│** HI-TECH C PROFESSIONAL DEVELOPMENT SYSTEM (Z80/180/64180 V7.60) **│**

**│** Copyright (C) 1984-1996 HI-TECH Software, All Rights Reserved **│**

**│** Serial no: CZ80-12345; Licensed to: **│**

**│** Clyde **│**

**│ │**

**└──────────────────────────────────────────────────────────────────────────────┘**

При запуске HPDZ ищет файлы TBL в текущем каталоге и каталоге справки, добавляя их в Меню справки. Путь каталога справки может быть определен переменной окружения HT\_xx\_HLP. Если она не задана, он будут получен из имени полного пути, используемого при вызове HPDZ. Если каталог справки не может быть найден, ни одна из стандартных записей справки не будет доступна.

**HPDZ**

Эта опция выводит окно со всеми темами, для которых доступна помощь. К ним относятся спецификации контрольной суммы, оптимизации компилятора, поиск в редакторе, модели памяти и типы микросхем, адреса ПЗУ и ОЗУ и поиск строк.

**C Library Reference** (Справочник по библиотеке C)

Эта команда выбирает интерактивное руководство по стандартной библиотеке ANSI C. Будет представлено окно с индексом для руководства. Темы могут быть выбраны двойным щелчком мышью на них, или переместив курсор с помощью клавиш со стрелками и нажав Return.

Как только тема будет выбрана, содержимое окна изменится на запись для этой темы в отдельном окне. Вы можете переместиться по ссылке, используя клавиши управления курсором клавиатуры, и индекс может быть повторно введен, используя кнопку INDEX в нижней части окна.

При наличии мыши, можете следовать по гипертекстовым ссылкам двойным щелчком мышью на любом слове. Например, если вы будете в printf записи и двойном щелчке по ссылке на fprintf, Вы перейдете к записи для fprintf.

Это окно может быть изменено и перемещено по желанию, и таким образом может быть оставлено на экране, во время редактирования.

**Editor Keys** (Горячие кнопки редактора)

Эта опция отображает список команд редактора и соответствующие клавиши, активирующие эту команду.

**Technical Support** (Техническая поддержка)

Эта опция отображает список дилеров и их телефонные номера которыми можно воспользоваться для получения необходимой технической поддержки.

**ZC Compiler Options** (Опции компилятора ZC)

Эта опция отображает окно, показывающее все опции компилятора ZC. Они отображаются в таблице, показывающей опцию и ее назначение. Таблицу можно просматривать путем прокрутки, используя стандартные клавиши прокрутки или мышь.

**Z80/Z180 Instruction Set** (Набор инструкций Z80/Z180)

Эта опция отображает таблицу всего набора команд процессоров Z80/Z180.

**Z180/64180 I/O Registers Set** (Регистры ввода-вывода Z180/64180)

Эта опция отображает таблицу регистров ввода-вывода процессоров Z180 и 64180.

**Release notes** (Примечания к выпуску)

Эта опция отображает информацию о версии вашей программы. Вы можете просмотреть окно путем прокрутки, используя обычные клавиши прокрутки или мышь.

# Драйвер компилятора командной строки ZC

Драйвер ZC вызывается из командной строки DOS для компиляции и/или компоновки C программ. Если вы предпочитаете использовать интегрированную среду, смотрите [Раздел 3](#_Использование_HPDZ_2) "*Использование HPDZ*". ZC имеет следующий основной формат команды:

**ZC [опции] файлы [библиотеки]**

Обычно **опции** (обозначенные предшествующим тире '-') указываются перед именами файлов, но на самом деле это не существенно. Опции описываются ниже. **Файлы** могут представлять собой смесь исходных файлов (C или ассемблер) и объектных файлов. Порядок файлов не имеет значения, за исключением того, что он влияет на порядок, в котором код или данные появляются в памяти. **Библиотеки** - это список имен библиотек или параметров *-L* (см. страницу 84). Исходные файлы, объектные файлы и файлы библиотек ZC отличаются только типом или расширением файла. Распознаваемые типы файлов перечислены в [Таблице 4-1](#t41) на странице 74. Это означает, например, что файл ассемблера должен всегда иметь тип файла *.AS* (регистр букв не важен).

Таблица 4‑1 Типы файлов ZC

|  |  |
| --- | --- |
| **Тип файла** | **Значение** |
| **.C** | Исходный файл C |
| **.AS** | Исходный файл ассемблера |
| **.OBJ** | Файл с объектным кодом |
| **.LIB** | Файл объектных библиотек |

ZC проверит каждый параметр файла и выполнит соответствующие действия. C файлы будут скомпилированы, ассемблерные файлы будут ассемблированы. В конце, если не будет подавлено одной из опций, описанных позже, все объектные файлы, полученные в результате компиляции, ассемблирования или явно перечисленные, будут соединены с любыми указанными библиотеками. Функции из библиотек будут связаны только при наличии ссылки на них.

Вызов ZC только с объектными файлами в качестве аргументов (то есть без исходных файлов) будет означать выполнение только этапа компоновки. В Make-файлах типично использовать ZC с опцией -C для компиляции нескольких исходных файлов в объектные файлы, а затем для создания окончательной программы, вызывать ZC только с объектными файлами и библиотеками (и соответствующими опциями).

### Длинные командные строки

Так как командная строка DOS ограничена 128 символами, для вызова ZC с длинным списком опций и файлов, можно создать командный файл, содержащий командную строку ZC, и вызвать ZC с вводом, перенаправленным ему из этого файла. Без указанных опций командной строки ZC использует стандартный ввод для получения списка своих аргументов. Например, командный файл может содержать:

**-v -O -Otest.hex -A0,8000,8000 -bl**

**file1.obj file2.obj mylib.lib**

Если это файл *xyz.cmd*, то ZC вызывается следующим образом:

**ZC < xyz.cmd**

Поскольку аргументы командной строки не указаны, ZC считает *xyz.cmd* в качестве своей командной строки.

Таблица 4‑2 Опции ZC

| **Символ** | **Назначение** |
| --- | --- |
| **-180** | Генерировать код для процессора Z180 |
| **-64180** | Генерировать код для процессора 64180 |
| **-A*spec*** | Указать адреса памяти для связывания |
| **-AAHEX** | Генерировать файл American Automation symbolic HEX |
| **-ALTREG** | Использовать альтернативный набор регистров |
| **-ASMLIST** | Генерировать файл ассемблера .LST для каждой компиляции |
| **-AV** | Выбрать формат таблицы символов Avocet |
| **-AVSIM** | Аналогичен -AV |
| **-BIN** | Генерировать двоичный выходной файл |
| **-Bs** | Выбор малой модели памяти |
| **-Bl** | Выбор большой модели памяти |
| **-Bc** | Выбор модели памяти CP/M |
| **-C** | Компилировать только объектные файлы |
| **-CLIST** | Генерировать листинг исходного файла C |
| **-CPM** | Генерировать исполняемый файл CP/M |
| **-CR*file*** | Генерировать листинг перекрестных ссылок |
| **-D*macro*** | Определить макрос препроцессора |
| **-E** | Использовать формат "редактора" для ошибок компилятора |
| **-E*file*** | Перенаправить ошибки компилятора в файл |
| **-E+*file*** | Добавить ошибки в файл |
| **-G*file*** | Генерировать таблицу символов уровня источника |
| **-H*file*** | Генерировать таблицу символов без номеров строк и пр. |
| **-HELP** | Распечатать сводку опций |
| **-I*path*** | Определить путь каталога для включаемых файлов |
| **-L*library*** | Определить библиотеку, которая будет отсканирована компоновщиком |
| **-L-*option*** | Определить - опцию, передаваемую непосредственно компоновщику |
| **-M*file*** | Создать файл распределения памяти |
| **-MOTOROLA** | Генерировать выходной HEX файл в формате Motorola S1/S9 |
| **-N*length*** | Установка длины идентификатора равным ***length*** (по умолчанию - 31 символ) |
| **-O** | Включить локальную оптимизацию |
| **-O*file*** | Определить имя выходного файла |
| **-OF** | Оптимизация по скорости |
| **-OMF51** | Произвести выходной файл OMF-51 |
| **-P** | Предварительная обработка ассемблерных файлов |
| **-P8** | Использовать 8 разрядную адресацию портов |
| **-P16** | Использовать 16 разрядную адресацию портов |
| **-PRE** | Создание исходного кода обработанного препроцессором |
| **-PROTO** | Сгенерировать прототипы функций |
| **-PSECTMAP** | Отобразить полное использование сегментов памяти после связывания |
| **-q** | Определить тихий режим |
| **-ROMDATA** | Оставить данные инициализации в ПЗУ |
| **-ROM*ranges*** | Определите диапазоны ПЗУ для кода |
| **-S** | Скомпилировать только в ассемблерные исходные файлы |
| **-SA** | Скомпилировать в исходные файлы ассемблера AVMAC Avocet |
| **-STRICT** | Включить строгое соответствие ключевых слов ANSI |
| **-TEK** | Генерировать выходной HEX файл в формате Tektronix |
| **-UBROF** | Генерировать выходной файл в формате UBROF |
| **-UNSIGNED** | Сделать символьный тип по умолчанию unsigned (без знака) |
| **-U*symbol*** | Не определять предопределенный символ препроцессора |
| **-V** | Многословный: отображать строки команд компилятора |
| **-W*level*** | Установить уровень предупреждений компилятора |
| **-X** | Исключить локальные символы из таблицы символов |
| **-Z180** | Генерировать код для процессора Z180 |
| **-Zg** | Включить глобальную оптимизацию в генераторе кода |

### Библиотеки по умолчанию

Драйвер ZC по умолчанию осуществляет поиск в стандартной библиотеке C. Это всегда происходит в конце, после любых указанных пользователем библиотек. Конкретная библиотека зависит от модели памяти. Стандартная библиотека содержит версию printf, поддерживающую только значения int. Если с помощью printf или sprintf или связанных функций требуется печатать длинные значения (long), необходимо указать опцию **-LL**. Она выполнит поиск в "длинной" библиотеке. Для поддержки printf с плавающей точкой используется опция **-LF**.

### Стандартный стартовый (Startoff) модуль запуска времени выполнения

Драйвер ZC также автоматически предоставит стандартный стартовый модуль времени выполнения, подходящий для модели памяти. Если вам требуется какая-либо специальная инициализация при включении питания (powerup), вместо замены или изменения стандартного модуля запуска (startoff) времени выполнения, вы должны использовать функции процедуры powerup (смотрите страницу 118).

## Опции компилятора ZC

Прежде всего, компилятор настроен для генерации кода ПЗУ, но опционально создаст исполняемые программы для операционной системы CP/M (с помощью опции -CPM). ZC распознает опции компилятора, перечисленные в [Таблице 4-2](#t42) на странице 75. Команда ZC также предоставляет доступ ко многим дополнительным функциям компилятора, которые не доступны в интегрированной среде разработки HPDZ.

### -180: генерация кода Z180/64180

Эта опция выбирает генерацию кода для процессоров Z180/64180 и определяет соответствующие библиотеки для того процессора.

### -64180: генерация кода Z180/64180

Эта опция аналогична опции -180.

### -Aspec: определение адресов ПЗУ и ОЗУ

Опция -A используется для задания адресов ПЗУ и ОЗУ, которые будут использоваться для соединения вашего кода в абсолютных адресах. Эта опция принимает следующую форму для маленькой модели Z80:

**-Arom,ram,ramsize,nvram**

и эта форма для других моделей и процессоров:

**-Arom,ram,ramsize,ramphys,banklogi,banksize,bankphys,nvram**

где:

|  |  |
| --- | --- |
| **rom** | адрес в ПЗУ, где должен быть запущен программный код. Практически для всех систем Z80 это будет адрес 0, наименьший доступный адрес ПЗУ. При компиляции кода, который должен быть загружен в ОЗУ с помощью отладчика LUCIFER, этот адрес будет началом загружаемой области ОЗУ. Область ПЗУ начинается с вектора сброса, за которым следуют любые другие вектора прерываний, которые были инициализированы, используя макросы обработки вектора прерывания, определенные в <intrpt.h>. |
| **ram** | начальный адрес ОЗУ. Для систем Z180 он должен быть адресом в первых 64 Кбайт, в которых ОЗУ должна быть отображена (т.е. *логический адрес ОЗУ*). |
| **ramsize** | размер в байтах доступного ОЗУ, с начального адреса ОЗУ |
| **ramphys** | физический (20-разрядный) адрес, по которому ОЗУ фактически находится в системе Z180. Чтобы использовать ОЗУ, оно должно быть отображено в нижней 64 Кбайт памяти. Это значение требуется для кода большой модели и для кода Z180, если адреса ОЗУ вне нижних 64 Кбайт. |
| **banklogi** | логический начальный адрес области, в которой находится банк, в пределах более низкого адресного пространства 64 Кбайт. Это и следующее значение, **banksize**, определяют окно в нижней части 64 Кбайт, в котором отображается ОЗУ из других физических адресов. |
| **banksize** | размер переключаемой области, в которой отображается ОЗУ из различных физических адресов. |
| **bankphys** | начало ПЗУ, которое должно быть отображено в переключаемую область |
| **nvram** | адрес блока энергонезависимого ОЗУ, используемого для переменных длительного хранения (persistent). Если все ОЗУ энергонезависимое, или переменные длительного хранения не используются, это значение должно быть равно нулю. Это приведет к размещению энергонезависимой области ОЗУ в стандартной области ОЗУ. Значение по умолчанию равно нулю. |

Для системы Z80, использующих малую модель, требуются только четыре значения (rom, ram, ramsize и nvram).

Обратите внимание, что для Z180 все логические и физические адреса должны быть кратны 1000H (4096 десятичных), потому что управление памятью Z180 выполняется 4096-байтными страницами.

Все значения, принимаемые опцией -A номера шестнадцатеричные (основание 16) числа и должны быть указаны без завершающего символа "H". Таким образом параметр

**-A0,2000,2000**

корректен, а опция:

**-A0H,2000H,2000H**

нет. Некоторые примеры допустимых опций -A показаны ниже:

Система Z80 с 32 Кбайт ПЗУ расположенным по адресу 0H; и 32 Кбайт ОЗУ, начинающимся по адресу 8000h потребует:

**-A0,8000,8000**

Система Z180 с ПЗУ 128 Кбайт, начинающимся по адресу 0 и 32 Кбайт ОЗУ, начинающимся по адресу 40000 (256 Кбайт) требует следующие опции:

**-A0,4000,8000,40000,1000,2000,4000**

В этом примере переключаемая область размещена с 1000 до 2FFF и отображается в физическую память, начиная с 4000. Так как 1000 шестнадцатеричных байтов ПЗУ, по-прежнему отображены с адреса 0, это оставит 3000 шестнадцатеричных байтов ПЗУ неиспользованными между этим и началом переключаемого ПЗУ.

### -AAHEX: Генерировать файл American Automation symbolic HEX

Опция -AAHEX указывает ZC генерировать файл в шестнадцатеричном символьном формате American Automation с расширением .HEX. Эта опция не действует, если используется с файлом .BIN. Шестнадцатеричный формат American Automation является расширением формата S-записей Motorola, который включает записи символов в начале файла. Эта опция должна использоваться при создании кода, который предназначен для отладки с помощью внутрисхемного эмулятора American Automation.

### -ALTREG: Использовать альтернативный набор регистров

Эта опция позволяет в обычных функциях использовать альтернативный набор регистров. Обратите внимание, что использование быстрого прерывания исключает использование альтернативного набора регистров любым другим способом, но использование в обычных функциях совместимо с существующим использованием альтернативного набора регистров в библиотечных функциях.

При возникновении конфликта возникает ошибка времени компоновки говорящая о том, что символ "быстрого прерывания не может быть использован с многократно определенными long, или float, или altreg". Это указывает на то, что были объявлены функции быстрого прерывания, но альтернативный набор регистров также используется для кода без прерывания.

Чтобы опция возымела эффект, также должна использоваться глобальная оптимизация (с помощью опции -Zg). С ней компилятор будет использовать альтернативный набор регистров для хранения переменных, когда это уместно.

### -ASMLIST: Генерировать файл ассемблера .LST

Опция -ASMLIST говорит ZC генерировать файл ассемблера .LST для каждой компиляции. Файл листинга показывает исходный C код и сгенерированный ассемблерный код и соответствующий двоичный код. Файл листинга будет иметь то же имя как исходный файл и тип файла (расширение) .LST.

Эта опция не совместима с опцией -CLIST.

### -AV: Выбрать формат таблицы символов Avocet

Опция -AV используется в сочетании с опцией -H для генерации таблицы символов в стиле Avocet для использования с эмулятором AVSIM и некоторыми внутрисхемными эмуляторами. Эта опция устанавливает только формат таблицы символов, она не дает команду компилятору генерировать файл символов. Для генерации файла символов Avocet, необходимо использовать опцию -AV с опцией -H. Например:

**ZC -AV -Htest.sym -A8000,8000 test.c**

будет генерировать таблицу символов в стиле Avocet, называемую test.sym. Опция -AV не должна использоваться с опцией ZC -G, так как таблицы символов Avocet не предусматривают отладочную информацию на уровне исходного кода.

### -AVSIM: Выбрать формат таблицы символов Avocet

Идентична опции -AV.

### -BIN: Генерировать двоичный выходной файл

Опция -BIN предписывает ZC генерировать выходной файл с бинарным образом. Выходной файл получит тип .BIN. Также двоичный выход может быть выбран, определив выходной файл типа .BIN в опции -Ofile.

### -Bs: Выбор малой модели памяти

Опция -Bs используется для выбора генерации кода для малой модели памяти. В малой модели используется адресное пространство 64 Кбайт. Библиотеки, используемые в малой модели, перечислены в [Таблице 4-3](#t43) на странице 79.

Таблица 4‑3 Библиотеки малой модели памяти

| **Библиотека** | **Назначение** |
| --- | --- |
| **RTZ80-S.OBJ** | Код запуска времени выполнения малой модели |
| **Z80-SC.LIB** | Стандартная библиотека малой модели |
| **Z80-SL.LIB** | Библиотека малой модели printf с поддержкой long |
| **Z80-SF.LIB** | Библиотека малой модели printf с поддержкой long и float |
| **RTZ801S.OBJ** | Z180 Код запуска времени выполнения малой модели |
| **Z801SC.LIB** | Z180 Стандартная библиотека малой модели |
| **Z801SL.LIB** | Z180 Библиотека малой модели printf с поддержкой long |
| **Z801SF.LIB** | Z180 Библиотека малой модели printf с поддержкой long и float |

### -Bl: Выбор большой модели памяти

Опция -Bl используется для генерации кода большой модели памяти, которая использует коммутацию банков памяти. Модуль времени выполнения начального запуска startoff и библиотеки, связанные с большой моделью, перечислены в [Таблице 4-4](#t44) на странице 80.

Таблица 4‑4 Библиотеки большой модели памяти

| **Библиотека** | **Назначение** |
| --- | --- |
| **RTZ80-L.OBJ** | Код запуска времени выполнения большой модели |
| **Z80-LC.LIB** | Стандартная библиотека большой модели |
| **Z80-LL.LIB** | Библиотека большой модели printf с поддержкой long |
| **Z80-LF.LIB** | Библиотека большой модели printf с поддержкой long и float |
| **RTZ801L.OBJ** | Z180 код запуска времени выполнения большой модели |
| **Z801LC.LIB** | Z180 стандартная библиотека большой модели |
| **Z801LL.LIB** | Z180 библиотека большой модели printf с поддержкой long |
| **Z801LF.LIB** | Z180 библиотека большой модели printf с поддержкой long и float |

Если большая модель используется с процессором Z180, переключение банков осуществляется с помощью Z180 MMU. Если большая модель используется с процессором, отличным от Z180, вы должны написать свои собственные процедуры коммутации банков. Дополнительные сведения смотрите в разделе [Соглашения о вызовах функций большой модели](#_Соглашения_о_вызовах) на странице 113.

### -Bc: Выбор модели памяти CP/M

Опция -Bc используется для генерации кода для модели памяти CP/M. Модель CP/M позволяет 64k адресное пространство и пользуется библиотеками CP/M. Модуль startoff и библиотеки времени выполнения, использующиеся со средней моделью перечислены в [Таблице 4-5](#t45) на странице 80. Обратите внимание, что опция -CPM эквивалентна опции -Bc.

Таблица 4‑5 Библиотеки модели памяти CP/M

| **Библиотека** | **Назначение** |
| --- | --- |
| **RTZ80-C.OBJ** | Код запуска startoff времени выполнения модели памяти CP/M |
| **Z80-CC.LIB** | Стандартная библиотека модели памяти CP/M |
| **Z80-CL.LIB** | Библиотека модели памяти CP/M printf с поддержкой long |
| **Z80-CF.LIB** | Библиотека модели памяти CP/M printf с поддержкой long и float |
| **RTZ801C.OBJ** | Z180 код запуска времени выполнения модели памяти CP/M |
| **Z801CC.LIB** | Z180 стандартная библиотека модели памяти CP/M |
| **Z801CL.LIB** | Z180 библиотека модели памяти CP/M printf с поддержкой long |
| **Z801CF.LIB** | Z180 библиотека модели памяти CP/M printf с поддержкой long и float |

### -C: Компилировать в объектные файлы

Опция -C используется, чтобы остановить компиляцию после генерации объектных файлов. Эта опция часто используется при компиляции нескольких исходных файлов, используя утилиту "make". Если компилятору указаны несколько исходных файлов, каждый из них будет скомпилирован в отдельный файл .OBJ. Чтобы скомпилировать три исходных файла main.c, module1.c и asmcode.as в объектные файлы можно использовать команду:

**ZC -O -Zg -C main.c module1.c asmcode.as**

Компилятор произведет три объектных файла main.obj, module1.obj и asmcode.obj которые затем можно соединить для создания шестнадцатеричного файла Motorola, используя команду:

**ZC -A0,8000,8000 main.obj module1.obj asmcode.obj**

Компилятор в командной строке примет любую комбинацию .C, .AS и .OBJ файлов. Исходные файлы на ассемблере будут переданы непосредственно ассемблеру и объектные файлы, не будут использоваться до вызова компоновщика. Если для определения имени выходного файла опция -O*file* не используется, в качестве окончательного вывода будет использоваться hex-файл Motorola с тем же "базовым именем", что и в первом исходном или объектном файле, в приведенном выше примере будет создан файл с именем main.hex.

### -CLIST: Генерировать листинг исходного файла C

Эта опция генерирует файл листинга для каждого исходного файла C, содержащие номера строк с символами табуляции шириной 8. Файл листинга называться file.LST, где file имеет базовое имя исходного файла C.

Эта опция не совместимая с опцией -ASMLIST.

### -CPM: Генерировать исполняемый файл CP/M

Опция -CPM эквивалентна опции -Bc и выбирает код и формат выходного файла CP/M.

### -CR*file*: Генерировать листинг перекрестных ссылок

Опция -CR произведет список перекрестных ссылок. Если параметр file опущен, "необработанная" информация о перекрестных ссылках будет оставлена во временном файле, после чего пользователь может запустит утилиту CREF. Если имя файла присутствует, например **-CRtest.crf**, ZC вызовет CREF для обработки информации о перекрестных ссылках в файл листинга, в этом случае TEST.CRF. Если несколько исходных файлов должны быть включены в список перекрестных ссылок, они все должны быть скомпилированы и соединены с помощью одной команды ZC. Например, чтобы сгенерировать список перекрестных ссылок, который включает исходные модули main.c, module1.c и nvram.c, выполните компиляцию и компоновку используя команду:

**ZC -CRmain.crf main.c module1.c nvram.c**

### -D*macro*: Определить макрос препроцессора

Опция -D используется, чтобы определить макрос препроцессора в командной строке, точно так, как если бы он был определен с помощью директивы #define в исходном коде. Эта опция может принять одну из двух форм, -Dmacro, которая эквивалентна:

**#define macro 1**

или -Dmacro=text, которая эквивалентна следующей:

**#define macro text**

Таким образом, команда:

**ZC -Ddebug -Dbuffers=10 test.c**

скомпилирует test.c с макросами, определенными точно так же, как если бы исходный код C включал директивы:

**#define debug 1**

**#define buffers 10**

### -E: Использовать формат "редактора" для ошибок компилятора

Опция -E дает компилятору команду генерировать сообщения об ошибках в формате, который приемлем для некоторых текстовых редакторов. Поведение по умолчанию, если -E не используется, состоит в том, чтобы отобразить ошибки компилятора в строке в "удобном для восприятия" формате с символом ^ и сообщением об ошибке, указывающим на некорректные символы в исходной строке, например:

**x.c: main()**

**4: PORT\_A = xFF;**

**^ undefined identifier: xFF**

Стандартный формат вполне приемлем для человека, читающего вывод ошибок, но не может использоваться с редакторами, которые поддерживают обработку ошибок компилятора. Если один и тот же исходный код скомпилирован с использованием параметра -E, вывод ошибок будет следующим:

**x.c 4 9: undefined identifier: xFF**

указывает, что ошибка произошла в файле x.c в строке 4, со смещением 9 символов в операторе. Второе числовое значение, номер столбца, относительно крайнего левого символа отличного от пробела в исходной строке. Если бы дополнительные пробелы или символ табуляции были вставлены в начале исходной строки, компилятор все равно сообщил бы об ошибке в строке 4, столбце 9. Вывод ошибок, в стандартном или в формате -E, может быть перенаправлен в файлы, используя UNIX, или DOS стандартный стиль перенаправление вывода. Ошибка из примера выше может быть перенаправлена в файл с названием errlist с помощью команды:

**ZC -E x.c > errlist**

Ошибки компилятора могут также быть добавлены в существующие файлы, используя синтаксис перенаправления и добавления. Если файл ошибок отсутствует, он будет создан. Для добавления ошибок компилятора в файл используется команда:

**ZC -E x.c >> errlist**

### -E*file*: Перенаправить ошибки компилятора в файл

Некоторые редакторы не позволяют использовать стандартные средства перенаправления командной строки при вызове компилятора. Чтобы работать с этими редакторами, ZC позволяет указывать имя файла листинга ошибок как часть опции -E. Файлы ошибок, сгенерированные с использованием этой опции, всегда будут иметь формат -E. Например, для компиляции x.c и перенаправления всех ошибок в x.err, используйте команду:

**ZC -Ex.err x.c**

Опция -E также позволяет добавлять ошибки в существующий файл, определяя + в начале имени файла ошибок, например:

**ZC -E+x.err y.c**

Если необходимо скомпилировать несколько файлов и объединить все сгенерированные ошибки в один текстовый файл, используйте опцию -E для создания файла, затем используйте -E+ при компиляции всех остальных исходных файлов.

Например, компиляция нескольких файлов с объединением всех ошибок в файл, названный project.err, можно использовать опцию -E следующим образом:

**ZC -Eproject.err -O -Zg -C main.c**

**ZC -E+project.err -O -Zg -C part1.c**

**ZC -E+project.err -C asmcode.as**

Файл project.err будет содержать любые ошибки из main.c, после чего следуют ошибки из файлов part1.c и затем asmcode.as, например:

**main.c 11 22: ) expected**

**main.c 63 0: ; expected**

**part1.c 5 0: type redeclared**

**part1.c 5 0: argument list conflicts with prototype**

**asmcode.as 14 0: Syntax error**

**asmcode.as 355 0: Undefined symbol \_putint**

### -G*file*: Генерировать таблицу символов уровня источника

Опция -G генерирует файл символов исходного уровня для использования с отладчиками HI-TECH Software и средствами моделирования, такими как Lucifer. Если имя файла не указано, файл символа будет иметь то же самое "базовое имя", что и первый исходный или объектный файл, и расширение .SYM. Например, **-GTEST.SYM** генерирует файл символов под названием TEST.SYM. Файлы символов, сгенерированные с использованием опции -G, включают информацию об исходном уровне для использования с отладчиками исходного уровня.

Эта опция не должна использоваться в сочетании с таблицами символов стиля Avocet (ZC опция -AV), поскольку формат таблицы символов Avocet не предусматривает предоставление информации об отладке на уровне исходного кода. Обратите внимание, что все исходные файлы, для которых требуется отладка на уровне исходного кода, должны быть скомпилированы с опцией -G, например:

**ZC -G -C test.c**

**ZC -C module1.c**

**ZC -A0,30,2000 -Gtest.sym test.obj module1.obj**

будет включать информацию об отладке на уровне исходного кода только для **test.c** потому, что **module1.c** не был скомпилирован с опцией -G.

### -H*file*: Генерировать файл символов уровня ассемблера

Опция -H генерирует файл символа без информации об исходном уровне. Отладчики HI-TECH Software смогут выполнять отладку только на уровне ассемблера при использовании файлов символов, сгенерированных с использованием -H. Обычно файл символов, сгенерированный с использованием этой опции, будет таблицей символов формата HI-TECH Software, однако, если эта опция используется вместе с опцией -AV ZC, будет генерироваться таблица символов стиля Avocet. Таблицы символов Avocet используются некоторыми внутрисхемными эмуляторами.

### -HELP: Отображение справки

Опция -HELP отображает информацию об опциях ZC.

### -I*path*: Путь поиска включаемых файлов Include

Используйте опцию -I, чтобы определить дополнительный каталог, используемый при поиске заголовочных файлов, которые были включены, используя директиву #include. Опция -I может использоваться несколько раз, если поиск должен быть осуществлен в нескольких каталогах. После поиска в каталогах определенных пользователем будет продолжен поиск в каталоге, содержащем все стандартные заголовочные файлы. Например:

**ZC -C -Ic:\include -Id:\myapp\include test.c**

будет искать в каталогах **c:\include** и **d:\myapp\include** любые включенные заголовочные файлы, использующие угловые скобки.

### -L*library*: Просматриваемая библиотека

Опция -L используется для указания дополнительных библиотек, которые должны быть проверены компоновщиком. Библиотеки, заданные с помощью опции -L, просматривается до стандартной библиотеки C, что позволяет обращаться к альтернативным версиям стандартных библиотечных функций. Например, при использовании Z80 малой модели памяти версия printf() с плавающей запятой может быть связана со стандартной версией путем поиска в библиотеке z80-sf.lib используя опцию -Lf. Аргумент -L является ключевым словом библиотеки, к которому добавляются префикс стандартной библиотеки и суффикс .LIB. Префикс библиотеки зависит от используемого процессора и модели памяти. Префиксы стандартных библиотек показаны на [Рисунке 4-1](#r41) на странице 84, где: тип процессора - 'z80-' для Z80 или 'z801' для Z180. ***Модель*** - это «s» для малых, 'l' для больших и 'c' для CP/M. И ***тип*** ***библиотеки***: 'c' - стандартная библиотека, 'l' - для printf, поддерживающей long и 'f' для printf поддерживающей float и long. (Тип библиотеки - это то, что можно указать с помощью этой опции.)

Все библиотеки должны находиться в подкаталоге LIB в каталоге установки компилятора. К библиотекам из других каталогов можно получить только с помощью HPDZ или путем непосредственного вызова компоновщика. Полный набор библиотек и модулей запуска, поставляемых с компилятором приведены в Таблицах [4-3](#t43), [4-4](#t44) и [4-5](#t45).

Рисунок 4‑1 Префиксы и суффиксы библиотеки

*Префикс Суффикс*

┌────────**┴**────────┐ ┌─**┴**─┐

**\_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ \_\_\_ . LIB**

**▲ ▲ ▲ ▲ ▲ ▲**

**│ │ │ │ │ │**

**Тип процессора ─┴───┴───┴───┘ │ └─ Тип библиотеки (c, f, l)**

**(z80-, z801) └───── Модель (s, l, c)**

### -L-*option*: Определить опцию, передаваемую компоновщику

Опция -L также может использоваться для указания дополнительной опции "-", передаваемой непосредственно компоновщику ZC. Если сразу за -L следует любой текст, начинающийся с символа "-", текст будет передан непосредственно компоновщику без обработки ZC. Например, если определена опция **-L-FOO**, опция -FOO будет передана компоновщику, когда он будет вызван. Опция -L особенно полезна при связывании кода, который содержит дополнительные секции программы (или psect), как может быть в случае, если программа содержит ассемблерный код или код C, который использует директиву #pragma psect. Если бы опция -L не существовала, было бы необходимо вызывать компоновщик вручную или использовать опцию HPDZ для связывания кода, использующего дополнительные psect. Опция -L позволяет определить любые дополнительные psect просто при помощи дополнительной опции компоновщика -P. Чтобы дать практический пример, предположим, что код содержит переменные, которые были отображены в специальную область ОЗУ, используя дополнительную psect, названную xram. Чтобы связать эту новую psect по соответствующему адресу, все, что вам нужно сделать, это передать дополнительную опцию компоновщика -P, используя опцию -L. Например, если специальная область ОЗУ (psect xram) должна находиться по адресу 2000h, можно использовать опцию **-L-Pxram=2000h** ZC следующим образом:

**ZC -Bl -L-Pxram=2000h -A0,4000,4000 prog.c xram.c**

Одна часто используемая опция компоновщика -N, которая сортирует таблицу символов в файле распределения памяти (map) по имени, а не по адресу. Она передается в ZC как -L-N.

### -M*file*: Генерация файла распределения памяти

Опция -M используется, чтобы запросить генерацию файла распределения памяти (map). Если имя файла не определено, информация о распределении памяти отображается на экране, иначе будет использоваться имя файла, определенное в опции -M.

### -MOTOROLA: Генерировать выходной HEX файл в формате Motorola S1/S9

Опция -MOTOROLA предписывает ZC генерировать HEX файл с S-записями Motorola при создании файла с расширением .HEX. Эта опция не имеет никакого эффекта, если используется файл .BIN.

### -N*length*: Определить значимую длину идентификатора

По умолчанию идентификаторы усекаются до 31 символа, так что два идентификатора, совпадающие в своих первых 31 символах будет считаться идентичными. Это согласуется с минимумом 31 символ, определенным в стандарте ANSI/ISO для C. Некоторые приложения могут потребовать, чтобы идентификаторы различались более чем 31 символами. Эта опция определяет значимое число символов в длину. Оно не может быть установлено вне границ 31-255. Используйте эту опцию осторожно, так как использование идентификаторов длиннее 31 символа будет означать, что код не может компилироваться другими ANSI-совместимыми компиляторами.

### -O: Включить локальную оптимизацию

Опция -O вызывает локальную оптимизатор после прохода генерации кода. Локальная оптимизация уменьшает размер кода, удаляя избыточные переходы и инструкции загрузки регистров.

### -O*file*: Определить имя выходного файла

Эта опция позволяет указать компилятору имя и тип выходного файла. Если опция "-O" не определена, выходной файл получит имя первого источника или объектного файла. Опцию -O можно использовать для определения типа выходного файла: HEX, BIN или UBR, содержащие данные в HEX, двоичном или формате UBROF соответственно. Например:

**ZC -Otest.bin -A0,4000,4000 prog1.c part2.c**

создаст двоичный файл с именем **TEST.BIN**.

Эта опция не влияет на .obj или .as файлы, произведенные компилятором.

### -OF: Оптимизация по скорости

Эта опция оптимизирует код, сгенерированный компилятором для скорости.

### -OMF51: Произвести выходной файл OMF-51

Эта опция заставит компилятор генерировать выходной (исполняемый код) файл в формате Intel OMF-51 (строго говоря, AOMF-51). Этот формат используется некоторыми внутрисхемными эмуляторами. Он поддерживает адресное пространство только 64 Кбайт и для кода и для данных.

Эта опция вряд ли будет иметь любое применение для процессоров семейства Z80.

### -P: Предварительная обработка ассемблерных файлов

Опция -P вызывает предварительную обработку ассемблерных файлов, прежде чем они будут собраны. Это позволяет ассемблерным файлам использовать #include, #if и т.д.

### -P8: Использовать 8 разрядную адресацию портов

Системы Z80, использующие 8-разрядную адресацию портов, могут извлечь выгоду из использования прямых 8-разрядных инструкций IN и OUT. Эта опция позволяет использовать эти инструкции. По умолчанию используется 16-разрядная адресация портов. Для Z180 эта опция не требуется и не желательна. Опция Z180 автоматически заставляет компилятор использовать инструкции IN0 и OUT0.

### -P16: Использовать 16 разрядную адресацию портов

Это - значение по умолчанию. Смотрите опцию -P8 для получения дополнительной информации.

### -PRE: Создание исходного кода обработанного препроцессором

Опция -PRE используется для создания предварительно обработанных исходных файлов C с расширением .PRE. Используйте эту опцию при отправке исходного кода в службу технической поддержки.

### -PROTO: Сгенерировать прототипы функций

Опция -PROTO используется для создания файлов .PRO, содержащих объявления функций в стиле ANSI и K&R для всех функций в указанных исходных файлах. Каждый созданный файл .PRO будет иметь то же базовое имя, что и соответствующий исходный файл. Файлы-прототипы содержат как прототипы стиля ANSI C, так и старые объявления функций C в условных блоках компиляции. Объявления extern из каждого файла .PRO должны быть отредактированы в глобальный файл заголовков, который включается во все исходные файлы, содержащие проект. Файлы .PRO могут также содержать статические объявления для функций, которые являются локальными для исходного файла. Эти статические объявления должны быть отредактированы в начале исходного файла. Чтобы продемонстрировать работу опции -PROTO, введите в качестве файла test.c следующий исходный код:

**#include <stdio.h>**

**int add(arg1, arg2) int \* arg1; int \* arg2; {**

**return \*arg1 + \*arg2;**

**}**

**void printlist(list, count) int \* list; int count; {**

**while (count--)**

**printf("%d ", \*list++);**

**putchar('\n');**

**}**

Если программу скомпилировать с помощью команды **ZC** **-PROTO** **test.c**, ZC создаст файл **test.pro**, содержащий следующие объявления, которые затем могут быть отредактированы по мере необходимости:

**/\* Prototypes from test.c \*/**

**/\* extern functions - include these in a header file \*/**

**#if PROTOTYPES**

**extern int add(int \*, int);**

**extern void printlist(int \*, int);**

**#else /\* PROTOTYPES \*/**

**extern int add();**

**extern void printlist();**

**#endif /\* PROTOTYPES \*/**

### -PSECTMAP: Отобразить полное использование памяти

Опция -PSECTMAP используется для отображения полного дампа памяти и psect (программных секций) после связывания кода пользователя. Информация, предоставляемая этой опцией, является более подробной, чем стандартный вывод распределения памяти, обычно отображаемый после компоновки. Опция -PSECTMAP заставляет компилятор распечатывать список всех psect компилятора и пользователя, а затем стандартную карту использования памяти. Например:

Psect Usage Map:

Psect | Contents | Memory Range

--------|--------------------------|-------------------------

lowtext | User defined psect | 0071H - 00F3H

vectors | Interrupt vectors | 0000H - 0070H

text | Program and library code | 00F4H - 0D25H

strings | Unnamed string constants | 0D26H - 0D89H

const | 'const' class data | 0D8AH - 0EADH

bss | Uninitialized RAM vars | 4000H - 4025H

Memory Usage Map:

User: 0071H - 00F3H 0083H (131) bytes

CODE: 0000H - 0070H 0071H (113) bytes

CODE: 00F4H - 0EADH 0DBAH (3514) bytes

RAM: 4000H - 4025H 0026H (38) bytes

### -q: Тихий режим

Если используется, эта опция должна быть первой. Она переводит компилятор в тихий режим, который подавляет вывод уведомления об авторских правах HI-TECH Software.

### –ROMDATA: Оставить данные инициализации в ПЗУ

По умолчанию программа компонуется так, что инициализируемые данные (т.е. любые статические или глобальные переменные или массивы, которые статически инициализированы, но не квалифицированы в качестве const) в начале работы программы копируются из ПЗУ в ОЗУ. Это позволяет изменять данные во время выполнения в соответствии со стандартной практикой C.

Однако, часто предпочтительнее, чтобы инициализированные данные оставались в ПЗУ, тем самым экономя место в ОЗУ. Опция -ROMDATA выбирает эту альтернативу. Инициализированные данные не изменяются во время выполнения. Эта опция не влияет на строковые константы и константные (const) данные и всегда остаются в ПЗУ.

### –ROMranges: Определите диапазоны ПЗУ для кода

Код программы может быть размещен в определенные области памяти кода - векторы прерываний и некоторый другой код должны быть размещены в фиксированных адресах, и адрес ПЗУ, определенный в опции -A, всегда используется для этого. Если используется опция -ROM, то, размещения другого кода, выше выделенного векторам, он будет размещен в указанные диапазоны. Синтаксис идентичен - опции -RAM, например.

**-ROM0-2FFF,4000-5FFF**

Обратите внимание, что нет необходимости исключать области, используемые векторами из диапазонов ПЗУ, поскольку компоновщик сделает это автоматически.

### -S: Компиляция в ассемблерный код

Опция -S прекращает компиляцию после создания исходного файла ассемблера. Файл ассемблера будет сгенерирован для каждого исходного файла C, переданного в командной строке. Команда:

**ZC -O -Zg -S test.c**

произведет исходный ассемблерный файл, названный test.as, который содержит код, сгенерированный из test.c. Опции оптимизации -O и -Zg могут использоваться вместе с опцией -S, позволяя исследовать выход компилятора для любого заданного набора опций. Эта опция особенно полезна для проверки функциональных соглашений о вызовах и значений "сигнатуры" при попытке написания внешних подпрограмм на языке ассемблера.

### -SA: Компиляция в исходные файлы ассемблера AVMAC Avocet

Эта опция позволяет компилятору скомпилировать исходный код в исходные файлы ассемблера AVMAC Avocet.

### -STRICT: Включить строгое соответствие ANSI

Опция -STRICT используется, чтобы включить строгое соответствие ANSI всех специальных ключевых слов. HI-TECH C поддерживают различные специальные ключевые слова (например, port для типов данных портов ввода-вывода). Если используется опция -STRICT, эти ключевые слова изменяются, используя двойное подчеркивание вначале (например, \_\_port), с тем чтобы строго соответствовать стандарту ANSI. Помните, что использование этой опции может вызвать проблемы с некоторыми стандартными файлами заголовков (например, INTRPT.H).

### -TEK: Генерировать выходной HEX файл в формате Tektronix

Опция -TEK указывает компилятору создавать файл HEX в формате Tektronix, если файл имеет расширением .HEX. Этот параметр не действует, если используется расширение .BIN.

### -U*macro*: Отмена определения макроса

Опция -U является инверсией опции -D, используется для отмены предопределенных макросов. Эта опция принимает форму **-D*macro***. Например, чтобы удалить предопределенный макрос z80 используют опцию -Uz80.

### -UBROF: Генерировать выходной файл в формате UBROF

Опция -UBROF указывает компилятору создать выходной файл в формате UBROF, подходящем для использования с некоторыми внутрисхемными эмуляторами. Выходному файлу будет присвоено расширение .UBR. Вывод UBROF также может быть выбран путем указания типа выходного файла .UBR, используя опцию -O. Эта опция не действует, если используется файл с расширением .BIN.

### -UNSIGNED: Сделать символьный тип по умолчанию unsigned

Опция -UNSIGNED трактует тип char по умолчанию как unsigned char. Поведение компилятора по умолчанию состоит в том, чтобы считать все значения символов и переменных unsigned char как signed char, если они явно не объявлены или не приведены. Если используется опция -UNSIGNED, тип char по умолчанию становится unsigned char, и переменные необходимо явно объявлять signed char. Диапазон signed char от -128 до +127, и диапазон unsigned char от 0 до 255.

### -V: Многословная компиляция

-V "многословная" опция. Компилятор отобразит командную строку, используемую для вызова каждого из проходов компилятора. Эта опция может быть полезна для определения точных параметров компоновщика, которые следует использовать, если вы хотите напрямую вызвать команду HLINK.

### -W*level*: Установить уровень предупреждений

Опция -W используется для установки уровня предупреждений компилятора. Допустимые уровни предупреждений находятся в пределах от -9 до 9. Уровень предупреждения определяет, насколько придирчив компилятор к сомнительным преобразованиям типов и конструкциям. Уровень предупреждения -W0 является значением по умолчанию и позволит все обычные предупреждающие сообщения. Предупреждение уровня -W1 подавляют сообщение "Func() declared implicit int" (функция неявно объявлена int). -W3 рекомендуется для компиляции кода, первоначально написанного другим, менее строгим, компиляторами. -W9 подавит все предупреждающие сообщения. Отрицательные уровни предупреждения -W-1, -W-2 и -W-3 включают специальные предупреждающие сообщения, в том числе проверку во время компиляции соответствие аргументов в printf() определенной строке формата.

### -X: Исключить локальные символы

Опция -X удаляет локальные символы из любых скомпилированных, ассемблированных или скомпонованных файлов. В любых объектных файлах или файлах символов остаются только глобальные символы.

### -Z180: Генерировать код Z180

Эта опция идентична опции -180 и выбирает генерацию кода для процессора Z180 и указывает соответствующие библиотеки для этого процессора.

### -Zg: Глобальная оптимизация

Опция -Zg вызывает глобальную оптимизацию во время прохода генерации кода. Это может привести к значительному сокращению размера кода и использования ОЗУ.

# Функции и окружение времени выполнения

Компилятор HI-TECH C поддерживают много специальных функций и расширений языка C, которые разработаны, для упрощения создания приложений использующих ПЗУ. Эта глава описывает опции компилятора и доступные специальные функции. После прочтения и понимания этого руководства вы будете знать, как:

* Скомпилировать свои C и ассемблерные исходные файлы, используя команду ZC.
* Скомпоновать свое приложение C для определенных адресов и создать бинарные образы, HEX файлы, файлы UBROF или файлы символов, подходящие для использования программаторами ППЗУ, отладчиками, средствами моделирования и внутрисхемными эмуляторами.
* Сконфигурировать подпрограммы консольного ввода-вывода, чтобы вы могли использовать подпрограммы <stdio.h> на своих аппаратных средствах.
* Настроить векторы и обработчики прерываний, используя только код C.
* Программировать устройства ввода-вывода отображенные в памяти, используя только код C.
* Организовать интерфейс между C и ассемблерным кодом, используя встроенные или внешние подпрограммы ассемблера.
* Понять три модели памяти, используемые для выполнения C-кода на Z80, и ограничения, налагаемые на ваше приложение C.

## Форматы выходных файлов

Компилятор может непосредственно создавать выходные файлы нескольких форматов, которые используются обычными программаторами ППЗУ и внутрисхемными эмуляторами. Интегрированный драйвер компилятора среды HPDZ, позволяет выбрать формат: Motorola Hex, Intel Hex, Binary, UBROF[[14]](#footnote-14), Tektronix Hex, American Automation symbolic Hex, Intel OMF-51 или Bytecraft .COD, используя пункт меню "**Output file type**" в меню "**Options**". По умолчанию команда ZC производит вывод в формате Intel HEX. Если имя выходного файла или тип не определены, ZC произведет файл Intel Hex с базовым именем первого источника или объектного файла. [Таблица 5-1](#t51) на странице 91 показывает опции выходного формата, доступные в ZC. С любой из опций выходного формата базовое имя выходного файла совпадет с именем первого источника, или объектного файла переданного ZC.

Таблица 5‑1 Форматы выходных файлов

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя формата** | **Описание** | **Опция ZC** | **Тип файла** |
| Motorola HEX | Файл типа S1/S9 hex | **-MOTOROLA** | **.HEX** |
| Intel HEX | Записи в стиле Intel hex (по умолчанию) |  | **.HEX** |
| Binary | Простой двоичный образ | **-BIN** | **.BIN** |
| UBROF | Формат универсального двоичного перемещаемого образа | **-UBROF** | **.UBR** |
| Tektronix HEX | Записи в стиле Tektronix hex | **-TEK** | **.HEX** |
| American Automation HEX | Формат Hex с символами для эмуляторов American Automation | **-AAHEX** | **.HEX** |
| OMF-51 | Формат абсолютного объектного модуля Intel | **-OMF** | **.OMF** |
| Bytecraft .COD | Формат кода Bytecraft | н.д. | **.COD** |

Столбец "Тип файла" указывает расширение файла, используемое для выходного файла.

В дополнение к показанным опциям опция -O может использоваться, чтобы запросить генерацию двоичного файла или файлов UBROF. Если вы будете использовать опцию -O определите имя выходного файла с типом *.BIN*, например -Otest.bin, то ZC произведет двоичный файл. Аналогично, для создания файла UBROF, можно использовать опцию -O определив выходной файл с типом *.UBR*, например -Otest.ubr.

Обратите внимание, что формат кода Bytecraft имеет ограниченное использование для Z80.

## Файлы символов

ZC опции -G и -H предписывают компилятору создать файл символов, который может использоваться отладчиками и средствами моделирования, чтобы выполнить символьную и отладку на уровне исходного кода. Опция -H производит файлы символов, которые содержат информацию только на уровне ассемблера, тогда как опция -G также включает исходную информацию уровня C. Если имя файла символов не определено, то по умолчанию будет создан файл, названный *file.sym*, где file - базовое имя первого исходного файла в командной строке. Например, чтобы создать файл символов, названный test.sym, содержащий исходную информацию уровня C:

**ZC -Gtest.sym test.c**

Файлы символов, созданные этими опциями, могут использоваться с внутрисхемными эмуляторами, и также отладчиком Lucifer, поставляемым с компилятором.

### Таблицы символов Avocet

Опция ZC -AV может использоваться в сочетании с опцией -H для создания таблицы символов стиля Avocet для использования со средством моделирования AVSIM и определенными внутрисхемными эмуляторами. Опция -AV не должна использоваться с опцией -G, поскольку формат таблицы символов Avocet не поддерживает информацию для отладки на уровне исходного кода.

## Предопределенные макросы

Драйверы компилятора определяют некоторые символы для препроцессора (CPP), позволяя условную компиляцию на основе типа микросхемы, модели памяти и т.д. Определенные символы перечислены в [Таблице 5-2](#t52) на стр. 92. Каждый символ, если он определен, приравнен к 1.

Таблица 5‑2 Предопределенные символы CPP

|  |  |
| --- | --- |
| **Символ** | **Использование** |
| **HI\_TECH\_C** | Установлен всегда - может использоваться, чтобы указать, что используется компилятор HI-TECH C. |
| **z80** | Установлен всегда - может использоваться для указания, что код скомпилирован для семейства процессоров z80. Обратите внимание, что этот символ в нижнем регистре. |
| **SMALL\_MODEL** | Установлен для малой модели памяти |
| **LARGE\_MODEL** | Установлен для большой модели памяти |
| **CPM** | Установлен для модели CP/M |
| **\_HOSTED** | Установлен при выполнении под операционной системой, т.е. при использовании модели CP/M |

## Поддерживаемые типы данных

Компилятор ZC поддерживает базовые типы данных размером 1, 2 и 4 байта. Все многобайтовые типы используют формат в котором сначала следует наименее значащий байт, также известный как *прямой порядок байтов* (little endian). Значения размером в слово, таким образом, хранят наименее значимый байт в ячейке с наименьшим адресом, а значения размером в двойное слово наименее значимый байт и наименее значимое слово хранят в ячейке с наименьшим адресом.

### 8 разрядные целочисленные типы данных

Компилятор HI-TECH C поддерживает signed char (со знаком) и unsigned char (без знака) 8 разрядные целочисленные типы. Тип char по умолчанию является signed char, если не используется ZC опция -UNSIGNED, в этом случае он является unsigned char. Тип signed char представляет собой 8 разрядное дополненное до 2 целое, представляющее целочисленные значения от -128 до +127 включительно. Тип unsigned char 8 разрядное целое без знака, представляющее целочисленные значения от 0 до 255 включительно. Существует распространенное заблуждение, что типы char в языке C предназначены исключительно для обработки символов ASCII. Это не так, ведь язык C не дает гарантий, что по умолчанию представление символов - числа ASCII. Типы char являются просто самыми меньшим из четырех возможных целочисленных размеров и ведут себя во всех отношениях как целые числа. Причина названия char историческая и не означает, что только char может использоваться для представления символов. В выражениях C можно свободно смешивать значения char с short, int и long. На Z80 типы char обычно используются во многих целях, как 8 разрядные целые числа, для хранения символов ASCII, и для доступа к расположениям ввода-вывода. Тип unsigned char является самым эффективным типом данных для Z80 и отображается непосредственно на 8 разрядные байты, которыми наиболее эффективно управляют инструкции Z80. Целесообразно, везде где возможно использовать типы char, чтобы увеличить производительность и уменьшить размер кода.

### 16 разрядные целочисленные типы данных

Компилятор HI-TECH C поддерживает четыре 16 разрядных целочисленных типа. Типы int и short представляет собой 16 разрядное дополненное до 2 целое со знаком, представляющее целочисленные значения от -32768 до +32767 включительно. Типы unsigned int и unsigned short - 16 разрядные целые без знака, представляющее целочисленные значения от 0 до 65535 включительно. 16 разрядные целочисленные значения представляются в формате с прямым порядком байтов с наименее значимым байтом по младшему адресу. Типы int и short являются 16 разрядными, и это самый маленький целочисленный размер, позволенный стандартом ANSI C. Таким образом, 16 разрядное представление int было выбрано, чтобы не нарушить стандарт ANSI. Использование целого меньшего размера, например 8 разрядного привело бы к серьезной несовместимости со стандартом C. Целые 8 разрядные числа полностью поддерживаются типами char и по мере возможности должны использоваться вместе с int.

### 32 разрядные целочисленные типы данных

Компилятор HI-TECH C поддерживает два 32 разрядных целочисленных типа. Тип long является 32 разрядным целым дополненным до 2, представляющим целочисленные значения от -2147483648 до +2147483647 включительно. Тип unsigned long является 32 разрядным целым без знака, представляющим целочисленные значения от 0 до 4294967295 включительно. Целочисленные 32 разрядные значения представляются в формате с прямым порядком байтов с младшим значащим словом и младшим значащим байтом в самом низком адресе. Типы long и unsigned long используют 32 бита, поскольку это наименьший размер длинного целого, допустимый в стандарте ANSI C.

### Числа с плавающей точкой

Числа с плавающей точкой реализованы, используя 32 бита, отформатированные следующим образом:

Таблица 5‑3 Формат 32 разрядного числа с плавающей точкой

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 | 30 | 24 | 23 | 0 |
| Знак – 1 бит | Экспонента - 7 битов | | Мантисса – 24 бита | |

Байт, содержащий знак и восемь битов экспоненты, располагается по самому высокому адресу. Высокого уровня байт состоит из знакового бита (бит 7), и экспоненты -7 битного дополнения до 64, превышающие 64 (т.е. значение экспоненты 0 сохраняется как 64). Остающиеся 3 байта - 24 разрядная мантисса, в двоичной дополнительной форме. Число с плавающей точкой всегда нормализуется.

Doubles являются точно таким же, как floats.

## Абсолютные переменные

Глобальная или статическая переменная может быть расположена по абсолютному адресу следуя своей декларации с помощью конструкции @ address, например:

**volatile unsigned char Portvar @ 0x1020;**

объявит переменную, названную **Portvar**, расположенную по адресу 1020h. Обратите внимание, что компилятор не резервирует место, а просто приравнивает переменную к этому адресу, сгенерированный компилятором ассемблер будет включать строку вида:

**\_Portvar equ 1020h**

Обратите внимание, что компилятор и компоновщик не осуществляют проверок перекрытия абсолютных переменных с другими переменными, таким образом, программист обязан гарантировать, что абсолютные переменные выделяются только в памяти не в используемой для других целей.

Эта конструкция, прежде всего, предназначена для приравнивания адреса идентификатора C с регистром микропроцессора. Чтобы разместить определяемую пользователем переменную по абсолютному адресу, определите ее в отдельной psect и дайте компоновщику команду помещать эту psect по требуемому адресу. См. [Раздел 5.26.3](#Директива ) "Директива #pragma psect" на странице 122.

## Квалификатор типа port

К портам ввода-вывода Z80 можно получить доступ непосредственно с помощью квалификатора типа **port**. Например, рассмотрим следующее объявление:

**port unsigned char \* pptr;**

**port unsigned char io\_port @ 0xE0;**

Переменная pptr является указателем на 8-разрядный порт и переменная io\_port отображается непосредственно на порт 0E0H и будет доступна с использованием надлежащих инструкций IN и OUT. Например, операторы

**io\_port |= 0x40;**

**\*pptr = 0x10;**

сгенерируют следующий код:

**;z.c: 8: io\_port |= 0x40;**

**IN A,(0E0H)**

**OR 40H**

**OUT (0E0H),A**

**;z.c: 9: \*pptr = 0x10;**

**LD BC,(\_pptr)**

**LD A,10H**

**OUT (C),A**

Компилятор позволяет в адресах выше 0FFh использовать порты ввода-вывода. Опции -P8 и -P16 ZC используются для выбора генерации кода для 8 или 16 разрядных адресов порта. Опция -P8 используется в традиционных системах Z80, в котором декодируется только 8 битов адресного пространства ввода-вывода. Опция -P16 (является значением по умолчанию) используется при генерации кода для систем Z80, которые декодируют более 8 адресных шин. Использование переключателя -P16 вынудит использовать инструкции IN A,(C) и OUT (C),A, для всего ввода-вывода, поскольку инструкции в форме IN и OUT не должны использоваться с декодированием 16 разрядных портов. Если вы генерируете код Z180 (с переключателем -180 в ZC), компилятор может также использовать инструкции IN0 и OUT0, где это требуется, и нет необходимости использовать -P16 или -P8 в сочетании с 180.

## Структуры и объединения

Компилятор HI-TECH C поддерживает типы struct и union любого размера от одного байта и более. Структуры и объединения могут свободно передаваться в качестве аргумента функции и возвращаемого значения. Полностью поддерживаются указатели на структуры и объединения.

### Битовые поля в структурах

Компилятор HI-TECH C полностью поддерживают *битовые поля* (bit fields) в структурах. Битовые поля выделяются, начиная с младшего значащего бита. Битовые поля выделяются в 16 разрядных словах, первый выделенный бит будет младшим значащим битом младшего значащего байта слова. Битовые поля всегда выделяются модулями по 16 битов, начиная со старшего значащего бита. Когда объявлено битовое поле, оно выделяется в текущем модуле из 16 битов, если он подходит, иначе в структуре выделяется новое слово на 16 битов. Битовые поля никогда не пересекают границу между 16 разрядными словами, но могут охватить границу байта в данном модуле выделения из 16 битов. Например, объявление:

**struct {**

**unsigned hi : 1;**

**unsigned dummy : 14;**

**unsigned lo : 1;**

**} foo @ 0x10;**

произведет структуру, занимающую 2 байта с адреса 10h. Поле **hi** будет битом 0 из адреса 10h, **lo** будет битом 7 по адресу 11h. Младший значащий бит **dummy** будет битом 1 из адреса 10h и старший значащий бит 10h будет битом 6 по адресу 11h. Если битовое поле объявлено в структуре, которой назначен абсолютный адрес, память не будет выделена, а тот факт, что зарезервированы 16 битов, не важен, поэтому для моделирования байтовой позиции с битовыми полями вы можете просто определить биты как будто был занят только один байт.

Безымянные битовые поля могут быть объявлены, чтобы дополнить неиспользуемое пространство между активными битами в регистрах команд. Например, если **dummy** не используется структура выше, может быть объявлена так:

**struct {**

**unsigned hi : 1;**

**unsigned : 14;**

**unsigned lo : 1;**

**} foo @ 0x10;**

## Квалификаторы const и volatile

Компилятор HI-TECH C поддерживает использование квалификаторов ANSI типа *const* и *volatile*. Квалификатор типа *const* используется, чтобы сообщить компилятору, что объект имеет постоянное значение и не будет изменен. Если будет предпринята попытка изменить объект, объявленный как *const*, компилятор выдаст предупреждение. Пользовательские объекты, объявленные как *const*, помещаются в специальной *psect*, называемой *const*. Например:

**const int version = 3;**

Квалификатор типа *volatile* используется для сообщения компилятору, что объект не может гарантировать сохранность своего значения между последовательными доступами. Это препятствует тому, чтобы оптимизатор устранил по-видимому избыточные ссылки на объекты, объявленные *volatile*, потому что это может изменить поведение программы. Все порты ввода-вывода и любые переменные, которые могут быть изменены подпрограммами обработки прерываний, должны быть объявлены *volatile*, например:

**volatile unsigned char TDR @ 0x1020;**

## Специальные квалификаторы типа

HI-TECH C поддерживают специальные квалификаторы типа, *persistent* и *code*, позволяющие пользователю управлять размещением статических и переменных класса *extern* в конкретные адресные пространства. Если используется ZC опция -STRICT, они вводят спецификатор, изменены на *\_\_persistent* и *\_\_code*. Эти квалификаторы типа могут также применяться к указателям. Эти квалификаторы типа не могут использоваться с переменными класса *auto*. При использовании с переменными, локальными для функции, они должны сочетаться с описателем класса памяти *static*. Вы не можете написать:

**void func(void)**

**{**

**persistent int intvar;**

**.. другой код ..**

**}**

потому что intvar имеет класс auto. Чтобы объявить intvar как постоянную переменную, локальную для функции test(), напишите:

**static persistent int intvar;**

### Квалификатор типа persistent

По умолчанию все явно не инициализированные переменные C сбрасываются в ноль при запуске. Это согласуется с определением языка C. Однако бывают случаи, когда желательно, чтобы некоторые данные сохранялись при сбросах или даже циклах (включения-выключения) питания. Квалификатор типа persistent используется для определения переменных, которые не должны очищаться при запуске. Кроме того, любые переменные persistent будут сохранены в другой области памяти отличной от других переменных, и этой области памяти может быть присвоен определенный адрес (с помощью ZC опции -A, или меню "Options/ROM and RAM addresses ..."). Таким образом, если имеется энергонезависимое ОЗУ небольшого размера, переменные persistent могут быть присвоены этой памяти. С другой стороны, если вся память энергонезависимая, вы можете принять решение выделить переменным persistent адреса вместе с назначенными компилятором другим переменным (но они по прежнему будут очищены). Одним из преимуществ явного присвоения адреса переменным persistent состоит в том, что он может остаться фиксированным, даже если вы изменяете программу, и другим переменным выделены иные адреса. Это позволит сохранить конфигурационную и другую информацию при обновлении прошивки (микро-программы в ПЗУ).

Есть некоторые библиотечные процедуры для [проверки и инициализации данных persistent](#f_persist_check) - смотрите страницу 286 для получения дополнительной информации и примера использования данных persistent.

### Квалификатор типа code

Квалификатор типа code работает только с большой моделью и используется для помещения инициализированных статических объектов в банк ПЗУ ассоциированный с code. Инициализированный статический объект будет расположен в *текущем банке ПЗУ*. Объекты, объявленные как code, должны быть статически инициализированы, например:

**static code char sometext[] = "abcdef";**

определит местоположение sometext[] в текущем банке ПЗУ. Очевидно, что оно будет доступно только для кода, исполняемого в этом банке или в общем коде, поэтому указатель на код данных не должен передаваться любым функциям вне текущего банка, включая библиотечные функции.

## Указатели

HI-TECH C поддерживают указатели нескольких различных классов, размером 16 и 32 бит. Класс указателя по умолчанию - это 16-разрядный указатель, адресующий адреса в пределах 64 Кбайт. 32 разрядный указатель является указателем только на функцию в большой модели.

### Совмещение квалификаторов типа и указателей

Квалификаторы типа const, volatile, persistent и code, могут также применяться к указателям, управляя поведением объекта, адресуемого указателем. При использовании этих квалификаторов с объявлениями указателей необходимо соблюдать осторожность, чтобы избежать путаницы относительно того, применяется ли квалификатор к указателю, или к объекту, адресуемому указателем. Правило следующие: если квалификатор типа слева от "\*" в объявлении указателя, он применяется к объекту, адресуемого указателем. Если квалификатор типа справа от "\*", он применяется к самой переменной указателя. Для иллюстрации примера использования типа volatile объявление:

**volatile char \* nptr;**

объявляет указатель на символ volatile. Квалификатор типа volatile применяется к объекту адресуемого указателем, так как он находится слева от "\*" в объявлении указателя.

Объявление

**char \* volatile ptr;**

ведет себя совершенно по-другому. Квалификатор типа справа от "\*" и, следовательно, относится к фактической переменной указателя ptr, а не объекту, адресуемому указателем. Наконец, объявление:

**volatile char \* volatile nnptr;**

генерирует указатель volatile на переменную volatile.

### Указатели code

Указатель code работает только с большой моделью и используется для указания инициализированных статических объектов, помещенных в банк ПЗУ ассоциированный с code. Инициализированный статический объект будет расположен в текущем банке ПЗУ и будет доступен только для кода, исполняемого в этом банке или в общего кода. Поэтому указатель на данные code не должен передаваться любым функциям вне текущего банка, включая библиотечные функции.

Указатели класса code на константы и переменные класса code часто используются для доступа к строковым константам, таким как меню и подсказки, помещенные в ПЗУ. Следующий код иллюстрирует этот метод:

**#include <conio.h>**

**static code charhello[] = "Hello, world\n";**

**static void**

**code\_puts(code char \* cptr)**

**{**

**char ch;**

**while (ch = \*cptr++)**

**putch(ch);**

**}**

**main()**

**{**

**code\_puts(hello);**

**}**

Использование констант и указателей code может уменьшить использование ОЗУ в большой модели памяти, в которой в зависимости от опции компиляции -ROMDATA инициализированные переменные копируются в ОЗУ.

### Указатели const

Указатели на const должны использоваться при косвенном доступе к объектам, которые были объявлены, используя спецификатор const. Указатели const ведут себя почти тем же способом как класс указателя по умолчанию в каждой модели памяти, c той лишь разницей, что компилятор запрещает попытки записи через указатель на const. Таким образом, учитывая объявление:

**const char \* cptr;**

оператор

**ch = \*cptr;**

является корректным, а оператор:

**\*cptr = ch;**

нет. В программе указатели const всегда обращаются к ПЗУ, потому что объекты объявленные const хранятся в ПЗУ.

## Обработка прерываний в C

Компилятор включает функции, позволяющие обрабатывать прерывания без написания кода на ассемблере. К функции может быть применен квалификатор *interrupt*, позволяющий вызывать ее непосредственно из аппаратного прерывания. Компилятор обрабатывает функции *interrupt* иначе, чем обычные функции, генерируя код для сохранения и восстановления любых используемых регистров и выхода с использованием команды RETI вместо команды RET в конце функции. (Если используется опция ZC -STRICT, ключевое слово interrupt становится *\_\_interrupt*. Везде, где это руководство ссылается на ключевое слово *interrupt*, воспринимайте его как *\_\_interrupt*, если используется -STRICT.)

Функция обработки прерываний должна быть объявлена с типом interrupt void и может не иметь параметров. Ее нельзя вызвать непосредственно из кода C, но сама она может вызывать другие функции, с учетом определенных ограничений. Ниже приведен пример функции обработки прерывания:

**long tick\_count;**

**void interrupt**

**tc\_int(void) {**

**++tick\_count;**

**}**

Способ настройки векторов прерывания зависит от того, используете ли вы векторы прерывания режима 0, 1 или 2. Обратите внимание, что NMI[[15]](#footnote-15) всегда обрабатывается так, как если бы это было прерывание режима 0 или 1, даже если вы используете режим 2 для других прерываний.

Таблица 5‑4 Макросы и функции поддержки прерываний

|  |  |  |
| --- | --- | --- |
| **Имя макроса** | **Описание** | **Пример** |
| **ei** | Разрешение прерываний | ei(); |
| **di** | Запрет прерываний | di(); |
| **im** | Выбор режима прерываний | im(2); |
| **set\_vector** | Инициализация вектора прерываний | set\_vector(BRKINT, brkint\_isr); |
| **ROM\_VECTOR** | Инициализация вектора прерываний в ПЗУ | ROM\_VECTOR(IV\_T0, handler); |
| **RAM\_VECTOR** | Инициализация вектора прерываний в ОЗУ | RAM\_VECTOR(IV\_T0, handler); |
| **CHANGE\_VECTOR** | Изменение вектора в ОЗУ | CHANGE\_VECTOR(IV\_T0, handler); |
| **READ\_RAM\_VECTOR** | Получение текущее содержимого вектора в ОЗУ | iptr = READ\_RAM\_VECTOR(IV\_T0); |

### Макросы обработки прерываний

Стандартный заголовочный файл <intrpt.h> содержит несколько макросов и функций, которые полезны при обработке прерываний с использованием кода C. Они перечислены в [Таблице 5-4](#t54) на странице 99.

#### Макросы ei() и di()

Макросы ei() и di() могут использоваться для отключения и включения маскируемых прерываний. Полезно отключить прерывания при инициализации или обслуживании устройств ввода-вывода. Макрос di() отключает прерывания, очищая флаги разрешения прерывания, используя инструкцию DI. Аналогично, ei() разрешает прерывания, устанавливая флаги разрешения прерывания, используя инструкцию EI.

#### Макрос im()

Для выбора режима прерывания предусмотрен макрос im. Этот макрос генерирует строку кода ассемблера для выбора соответствующего режима прерывания. Для im(2) также будет установлен регистр I. Он должен вызываться в основной процедуре перед включением прерываний. Аргумент im() - номер режима прерывания 0, 1 или 2.

Например,

**im(2);**

Выбирает режим прерываний 2.

#### Функция set\_vector()

Функция set\_vector() позволяет инициализировать вектор прерываний. Смотрите описание этой функции в [Разделе 10](#f_set_vector) "*Библиотечные функции*".

#### ROM\_VECTOR

Макросы ROM\_VECTOR, RAM\_VECTOR, CHANGE\_VECTOR и READ\_RAM\_VECTOR могут использоваться только со 2 режимом прерываний.

Макросы ROM\_VECTOR, RAM\_VECTOR и CHANGE\_VECTOR используются для создания "жёстко закодированной" таблицы векторов в ПЗУ, выровненной по границе 256 байтов. Старший байт начального адреса этого массива при запуске загружается в регистр I. Таблица имеет достаточный размер для задания самого высокого вектора.

Макрос ROM\_VECTOR статически инициализирует запись в этой таблице. Он принимает форму:

**ROM\_VECTOR(vector, handler);**

где **vector** - смещение вектора (то есть 8-разрядное значение, которое будет представлено устройством прерывания в ответ на подтверждение прерывания), а **handler** - имя функции, которая будет обрабатывать прерывание.

Макрос ROM\_VECTOR генерирует встроенный ассемблерный код, таким образом, параметр **vector** может быть в любом формате, приемлемом для ассемблера. Шестнадцатеричные адреса вектора прерываний могут быть переданы в шестнадцатеричном стиле языка C (0xA0) или ассемблера (A0h).

#### RAM\_VECTOR

Макрос RAM\_VECTOR устанавливает "программно-управляемый" вектор прерываний, который может быть изменен для указания на другой обработчик прерывания, если это необходимо. Обратите внимание, для того что бы это работало данные psect во время выполнения должны быть скопирована в ОЗУ. При использовании макроса RAM\_VECTOR не должна использоваться опция компилятора -ROMDATA, а также не может быть отключена опция HPDZ "Initialized data in RAM".

Как и в случае с ROM\_VECTOR, RAM\_VECTOR статически инициализирует запись в таблице векторов. RAM\_VECTOR принимает те же аргументы, что и ROM\_VECTOR, и может использоваться везде, где используется ROM\_VECTOR.

#### CHANGE\_VECTOR

Макрос CHANGE\_VECTOR используется для изменения вектора, который был установлен макросом RAM\_VECTOR. Это достигается путем изменения адреса обработчика прерываний во внутренней памяти. Например:

**di();**

**CHANGE\_VECTOR(0xA0, new\_handler);**

**ei();**

изменит адрес обработчика, использующего вектор 0xA0, чтобы указать на функцию обработки прерываний, названную new\_handler(). Адрес слова vector во внутренней памяти определяется специальным символом, определенным в RAM\_VECTOR, поэтому макрос CHANGE\_VECTOR следует использовать только в том же модуле и после использования RAM\_VECTOR. Адрес вектора должен быть идентичным, иначе символы не будут совпадать и возникнет ошибка времени компиляции.

Если вектор был, и необходимо изменить его обратно к исходному значению, нужно использовать CHANGE\_VECTOR, чтобы изменить его обратно. Повторное выполнение кода, который содержит макрос RAM\_VECTOR не сбросит вектор, потому что RAM\_VECTOR статически инициализирует вектор без генерации исполняемого кода. Единственным макросом инициализирующим вектор является CHANGE\_VECTOR, который генерирует инструкции реально выполняющиеся во время выполнения. Макросы ROM\_VECTOR и RAM\_VECTOR просто присваивают начальные значения векторам.

#### READ\_RAM\_VECTOR

Макрокоманда READ\_RAM\_VECTOR может использоваться для считывания значения вектора прерываний размещенного в ОЗУ, который был установлен с помощью макроса RAM\_VECTOR. Он никогда не должен использоваться с векторами, которые были инициализированы с помощью ROM\_VECTOR, поскольку будет возвращена бессмысленная информация. Макрос READ\_RAM\_VECTOR может использоваться совместно с CHANGE\_VECTOR для сохранения старого адреса обработчика прерываний, установки нового адреса и последующего восстановления исходного адреса. Например:

**volatile unsigned char wait\_flag;**

**interrupt void**

**wait\_handler(void)**

**{**

**++wait\_flag;**

**}**

**void**

**wait\_for\_serial\_intr(void)**

**{**

**interrupt void (\*old\_handler)(void);**

**di();**

**old\_handler = READ\_RAM\_VECTOR(0xA0);**

**wait\_flag = 0;**

**CHANGE\_VECTOR(0xA0, wait\_handler);**

**ei();**

**while (wait\_flag == 0)**

**continue;**

**di();**

**CHANGE\_VECTOR(0xA0, old\_handler);**

**ei();**

**}**

### Режимы прерываний

Режим прерываний 0 является совместимым с 8080, т.е. устройство прерывания, предоставляет единственную инструкцию (обычно JP или RST), в то время как режим прерываний 1 всегда вектор по адресу 0x38 (как инструкция **rst 38h**). Таким образом, в режимах прерываний 0 и 1 (и NMI) требуется инструкции JP или другой код, по адресу вектора.

С другой стороны, режим прерываний 2 выбирает 2-байтовый адрес из памяти в 256-байтовой странице векторов. Старший байт адреса вектора содержится во внутреннем регистре ЦП, и младший байт предоставляется устройством прерывания. Сформированный 2-байтовый адрес загружается в счетчик команд.

#### Настройка режима прерываний

Макрос im() генерирует строку ассемблерного кода для выбора соответствующего режима прерываний. Его необходимо вызвать в основной подпрограмме перед включением прерываний.

#### Прерывания в режиме 0 и режиме 1

Библиотечная подпрограмма set\_vector() предназначена для использования с режимами прерываний 0 или 1 и всегда должна использоваться для NMI, который всегда ожидает вектор в адресу 0x66. Подпрограмма set\_vector() зависит от кода времени выполнения startoff, устанавливающего инструкции JP в местах расположения вектора. Если вы хотите непосредственно настроить базовую страницу вектора прерываний для своих процедур обработки прерываний, можно изменить код времени выполнения startoff, чтобы непосредственно закодировать инструкции JP, которые указывают на ваш код. Векторы прерывания устанавливаются последовательностью кода, показанной ниже (из стандартного модуля startoff времени выполнения RTZ80-S.AS). Модуль времени выполнения startoff для кода с переключением банков и Z180 аналогичен этому.

**psect vectors**

**interrupt void**

**globalstart,\_main,\_exit,\_\_Hstack,\_\_Hbss, \_\_Lbss**

**globalpowerup, \_\_Ldata, \_\_Hdata, \_\_Lconst**

**defb 0c3h ; Код операции JP**

**defw init ; не может быть оптимизирован ZAS с помощью опции -J**

**defb 0,0**

**jp bdosvec ; переход к обработчику BDOS**

**jp r08vec ; переход к обработчику RST 08**

**defb 0,0,0,0,0**

**jp r10vec ; переход к обработчику RST 10**

**defb 0,0,0,0,0**

**jp r18vec ; переход к обработчику RST 18**

**defb 0,0,0,0,0**

**jp r20vec ; переход к обработчику RST 20**

**defb 0,0,0,0,0**

**jp r28vec ; переход к обработчику RST 28**

**defb 0**

**jp r2Cvec ; переход к обработчику RST 0C (NSC800)**

**defb 0**

**jp r30vec ; переход к обработчику RST 30**

**defb 0**

**jp r34vec ; переход к обработчику RST 0B (NSC800)**

**defb 0**

**jp r38vec ; переход к обработчику RST 38**

**defb 0**

**jp r3Cvec ; переход к обработчику RST 0A (NSC800)**

**defb 0,0**

**defm "Copyright (C) 1993 HI-TECH Software"**

**defb 0,0**

**jp nmivec** **; переход к обработчику NMI**

**;**

**fakesp:**

**defw start**

**init:**

**ld sp,fakesp**

**jp powerup**

Последовательность кода, показанная выше, скомпонована по адресу 0 в ПЗУ и перенаправляет все прерывания базовой страницы в таблицу инструкций JP, которые помещаются в ОЗУ с помощью кода начала выполнения (startoff). Процедура set\_vector() следует за инструкцией JP на базовой странице и изменяет инструкцию JP расположенную в ОЗУ на указанную процедуру. Например, вызов

**set\_vector((isr \*)0x38, handleint)**

разместит JP на handleint() в расположении ОЗУ, на которое указывает вектор RST 38, т.е. r38vec.

Этот метод позволяет динамически изменять векторы прерываний во время выполнения кода. Многие приложения, размещаемые в ПЗУ, используют только небольшое фиксированное количество векторов прерываний. Возможно, в этом случае целесообразно отредактировать модуль времени выполнения startoff и жестко закодировать базовую страницу векторов, чтобы указать непосредственно на подпрограммы прерывания. Например, вектор RST 38 может быть жестко закодирован, чтобы указать на handleint(), изменив его на:

**GLOBAL \_handleint**

**JP \_handleint ;38- RST 38 handler**

#### Прерывания в режиме 2

Режим прерывания 2 обрабатывается с помощью макросов ROM\_VECTOR, RAM\_VECTOR, CHANGE\_VECTOR и READ\_RAM\_VECTOR, как определено в стандартном заголовочном файле <intrpt.h>.

Ниже показан пример использование режима прерываний 2.

**#include <z180.h>**

**#include <intrpt.h>**

**#include <stdio.h>**

**#define CLOCK 6144000 /\* Тактовая частота в Гц \*/**

**#define TICKS 50 /\* Тактов в секунду \*/**

**#define COUNT ((CLOCK/20)/TICKS) /\* Отсчетов таймера за такт \*/**

**#define BRATE 0x01 /\* Скорость 19200 @ 6.144Mhz \*/**

**static unsigned time;**

**static unsigned char ticks, flag;**

**static void interrupt**

**timer\_int(void) {**

**if(++ticks == TICKS) {**

**ticks = 0;**

**time++;**

**flag = 1;**

**}**

**TCR; /\* Чтение TCR для очистки флага \*/**

**TMDR0L; /\* и регистров таймера \*/**

**}**

**void putch(char c) {**

**if(c == '\n') {**

**while (!(STAT0 & 2))**

**continue;**

**TDR0 = '\r';**

**}**

**while (!(STAT0 & 2)) /\* Ожидание TDRE == 1 \*/**

**continue;**

**TDR0 = c;**

**}**

**main(void) {**

**/\* Установка вектора прерываний \*/**

**ROM\_VECTOR(PRT0\_VEC, timer\_int);**

**im(2); /\* установка режима прерывания \*/**

**STAT0 = 0; /\* Сброс 1 канала ASCI \*/**

**CNTLA0 = 0x64; /\* Включение UART \*/**

**CNTLB0 = BRATE; /\* Скорость передачи \*/**

**RLDR0L = COUNT; /\* Установка таймера \*/**

**RLDR0H = COUNT >> 8;**

**TCR=0x11; /\* Запуск таймера,разрешение прерываний\*/**

**ei();**

**for(;;) {**

**while(!flag) /\* Ожидание несколько секунд! \*/**

**continue;**

**flag = 0;**

**printf("%6d\n", time); /\* Печать времени \*/**

**}**

**}**

### Предопределенные имена векторов прерываний

Заголовочный файл z180.h включает объявления для всех стандартных векторов прерываний для внутренних прерываний Z180. Эти имена векторов могут использоваться как аргумент адреса вектора для макросов ROM\_VECTOR, set\_vector(), RAM\_VECTOR, CHANGE\_VECTOR и READ\_RAM\_VECTOR.

Векторы прерывания, определенные в z180.h, перечислены в [Таблице 5-5](#t55) на странице 105. Вектора прерывания кроме тех что в z180.h могут быть объявлены, используя директивы препроцессора #define, или адрес вектора может непосредственно использоваться векторными макросами.

Таблица 5‑5 Вектора прерываний Z180

|  |  |  |
| --- | --- | --- |
| **Имя** | **Значение** | **Назначение** |
| **NT1\_VEC** | 0x00 | External /INT1 |
| **INT2\_VEC** | 0x02 | External /INT2 |
| **INTCAP\_VEC** | 0x12 | Input capture |
| **OUTCMP\_VEC** | 0x14 | Output compare |
| **TIMOV\_VEC** | 0x16 | Timer Overflow |
| **PRT0\_VEC** | 0x04 | PRT Channel 0 |
| **PRT1\_VEC** | 0x06 | PRT Channel 1 |
| **DMA0\_VEC** | 0x08 | DMA Channel 0 |
| **DMA1\_VEC** | 0x0A | DMA Channel 1 |
| **CSIO\_VEC** | 0x0C | Clocked serial I/O |
| **ASCI0\_VEC** | 0x0E | Async channel 0 |
| **ASCI1\_VEC** | 0x10 | Async channel 1 |

### Обработка немаскируемых прерываний

Для объявления функции обрабатывающей немаскируемое прерывание Z80 может использоваться классификатор функции nmi. Для выхода из функции, объявленной nmi interrupt используется инструкция retn вместо reti. Функции nmi interrupt должны быть векторизованы только из вектора NMI по адресу 66h, поскольку небезопасно завершить обычное прерывание с помощью retn или NMI с помощью reti, из-за способа обработки триггеров прерываний в Z80.

### Быстрые прерывания

Функция, объявленная fast interrupt (быстрое прерывание), сохраняет регистры общего назначения, переключая банки регистров, используя инструкции *EXX* и EX AF,AF'. Функции fast interrupt не могут использоваться в программах, которые используют арифметические возможности long (32 разряда) или с плавающей точкой в библиотеке C, поскольку некоторые библиотечные подпрограммы, использующие код с плавающей точкой, задействуют альтернативный набор регистров. Во времени компоновки производится проверка этого условия. Компоновщик завершает выполнение сообщением о соответствующей ошибке, если вы попытаетесь использовать быстрые прерывания и плавающую точку в одной и той же программе. Функции fast interrupt никогда не должны быть вложенными. На время выполнения функции прерывания должны быть отключены.

## Смешивание кода C и ассемблера Z80

Код языка ассемблера Z80 можно смешивать с кодом C тремя разными способами.

### Внешние функции на языке ассемблера

Функции могут быть полностью запрограммированы на ассемблере, ассемблированы ZAS как отдельные исходные файлы .AS, и объединены в бинарный образ, используя компоновщик. Этот метод позволяет передавать параметры и возвращаемые значения между C и ассемблерным кодом. Чтобы получить доступ к внешней функции, сначала включайте соответствующее объявление C extern в C код вызова. Например, предположим, что нужна функция языка ассемблера, чтобы обеспечить доступ к команде поворота влево на Z80:

**extern char rotate\_left(char);**

объявляет внешнюю функцию rotate\_left(), которая имеет тип возвращаемого значения char и принимает единственный аргумент типа char. Фактический код для rotate\_left() предоставлен внешним файлом .AS, который ассемблируется отдельно с помощью ZAS. Полный код rotate\_left() на ассемблере Z80 будет выглядеть примерно так:

**GLOBAL \_rotate\_left**

**SIGNAT \_rotate\_left,4153**

**PSECT text**

**\_rotate\_left:**

**RLC E**

**LD L,E**

**RET**

Именем функции на ассемблере является имя, объявленное в C с добавлением подчеркивания. Псевдо-операция GLOBAL является ассемблерным эквивалентом ключевого слова C extern, и псевдо-операция SIGNAT используется, чтобы осуществить проверку соглашения о вызовах во время компоновки. Более подробно проверка сигнатуры и псевдо-операция SIGNAT описываются далее в этой главе. Обратите внимание, что для корректной работы функций ассемблера, они должны считаны в нужном месте для любого переданного аргумента, и должны правильно настроить любые возвращаемые значения. Механизмы распределения локальных переменных, передачи аргументов и возвращаемых значений подробно рассматриваются в руководстве далее и должны быть изучены прежде чем пытаться писать процедуры на языке ассемблера.

### #asm, #endasm и asm()

Инструкции Z80 также могут быть непосредственно встроены в код C, используя директивы #asm, #endasm и asm(). Директивы #asm и #endasm используются для обозначения начала и конца блока ассемблерных инструкций, встраиваемых в код C. Директива asm() используется для размещения одной ассемблерной инструкции в код, генерируемый компилятором C. Чтобы продолжать наш пример выше, вы можете напрямую закодировать поворот влево в байте памяти, используя любой метод, показанный в следующем примере:

**#include <stdio.h>**

**unsigned char var;**

**main() {**

**var = 1;**

**printf("var = 0x%2.2X\n", var);**

**#asm**

**LD A,(\_var)**

**RLC A**

**LD (\_var),A**

**printf("var = 0x%2.2X\n", var);**

**asm("LD A,(\_var)");**

**asm("RLC A");**

**asm("LD (\_var),A");**

**printf("var = 0x%2.2X\n", var);**

**}**

При использовании встроенного кода ассемблера следует проявлять большую осторожность, чтобы избежать взаимодействия с кодом, генерируемым компилятором. Если есть сомнения, скомпилируйте вашу программу с помощью опции ZC -S и изучите код ассемблера, сгенерированный компилятором.

*Важное примечание*: конструкция #asm и #endasm не являются синтаксической частью программы C, и поэтому они не подчиняется обычным правилам изменения выполнения программы C. Например, вы не можете использовать блок #asm с оператором if и ожидать, что он будет работать правильно. Если вы используете встроенный ассемблер вместе с какими-либо конструкциями C, например if, while, do и т.д., необходимо использовать только форму asm(""), которая является оператором C и будет правильно взаимодействовать со всеми операторами управления выполнением C.

## Проверка сигнатуры

Компилятор автоматически создает сигнатуры для всех функций. *Сигнатура* представляет собой 16-разрядное значение, вычисленное из комбинации типа возвращаемого значения функции, количества параметров и другой информации, влияющей на последовательность вызова функции. Эта сигнатура выводится в объектном коде любой функции, ссылающейся или определяющей функцию.

Во время сборки компоновщик сообщит о любом несоответствии сигнатур. Таким образом, если функция будет объявлена в одном модуле иначе (например, как char вместо short), компоновщик сообщит об ошибке.

Иногда необходимо писать подпрограммы на языке ассемблера, которые вызывают из C, используя объявление extern. Такие функции языка ассемблера должны включать подпись, которая совместима с прототипом С, используемым для их вызова. Самый простой способ определения правильной сигнатуры для функции - написать фиктивную функцию C с тем же прототипом и скомпилировать ее на языке ассемблера, используя опцию ZC -S. Например, предположим, что у вас есть подпрограмма языка ассемблера с именем \_widget, которая принимает два аргумента int и возвращает значение char. Прототип, используемый для вызова этой функции из C, будет:

**extern char widget(int, int);**

Когда вызов \_widget выполняется в коде C, будет генерироваться сигнатура для функции с двумя аргументами int и значением возврата char. Чтобы соответствовать правильной сигнатуре, исходный код для widget должен содержать псевдо-оператор ZAS SIGNAT, который определяет одно и то же значение сигнатуры. Чтобы определить правильное значение, вы должны написать следующий код:

**char widget(int arg1, int arg2)**

**{**

**}**

и скомпилировать его в код ассемблера, используя команду

**ZC -S x.c**

Полученный код ассемблера включает следующую строку:

**signat \_widget,8249**

Псевдо-операция SIGNAT сообщает ассемблеру включить в файл .OBJ запись со значением 8249 символа \_widget. Значение 8249 является корректной сигнатурой функции с двумя аргументами int и возвращаемым значением char. Если эта строка будет скопирована в .AS файл, где определен \_widget, она свяжет корректную сигнатуру с функцией, и компоновщик будет в состоянии проверить корректность передачи параметров. Например, если другой .C файл содержит объявление:

**extern char widget(long);**

будет создана другая сигнатура, и компоновщик сообщит о несоответствии сигнатур, предупреждая о существования возможного нарушения соглашений о вызовах.

## Компоновка программ

Компилятор автоматически вызовет компоновщик, если не требуется остановиться после создания кода ассемблера (опция ZC -S) или объектного кода (опция ZC -C). Чтобы указать адрес ОЗУ (для переменных и стека) и адрес ПЗУ (для кода и инициализированных констант), используйте параметр ZC -A. Если опция -A не используется, компилятор запросит соответствующие адреса и размеры ПЗУ и ОЗУ.

ZC и HPDZ по умолчанию генерируют файлы Intel hex. Если вы используете опцию -BIN или определите выходной файл с типом файла .BIN, используя опцию ZC -O, то компилятор сгенерирует двоичный образ вместо него. Файл будет содержать код, начинающийся с наименьшего инициализированного адреса в программе. Например:

**ZC -v -oxx.bin -A0,8000,8000**

произведет двоичный файл, начинающийся с векторов, за которыми следует код пользователя, инициализированные данные и код библиотеки. После связывания компилятор автоматически сгенерирует карту использования памяти, которая показывает адрес и размер всех областей памяти, используемых скомпилированным кодом. Например:

**Memory Usage Map:**

**User: 0069H - 00F2H 008AH (138) bytes**

**ROM: 0000H - 0068H 0069H (105) bytes**

**ROM: 00F3H - 0DC2H 0CD0H (3280) bytes**

**RAM: FE00H - FEC6H 00C7H (199) bytes**

Более подробная информация об использовании памяти, перечисленная в порядке возрастания отдельных psect, может быть получена при помощи опции ZC -PSECTMAP.

## Использование памяти

Компилятор делает несколько предположений относительно памяти. За исключением переменных, объявленных с использованием конструкции @address, абсолютные адреса не выделяются до момента компоновки.

## Использование регистров

Регистр IX используется в качестве указателя стека, в то время как IY может использоваться в качестве регистровой переменной. Регистры DE и BC используются для передачи аргументов функции на основе регистров. Регистры HL и DE используются для возвращаемых значений функций. Любая вызываемая процедура на языке ассемблера должна сохранять эти регистры.

## Организация стекового кадра

При входе в \_main и все другие функции C, выполняется некоторый код для настройки локального стекового кадра. Он включает сохранение не временных регистров (IX и IY для Z80) и установку базового указателя (IX), на начало нового стекового кадра. IX и IY сохраняются, только если функция использует их. Затем указатель стека корректируется, чтобы обеспечить необходимое пространство для локальных переменных. Типичный код при входе в функцию следующий:

**PUSH IX**

**LD IX,0**

**ADD IX,SP**

**LD HL,-10**

**ADD HL,SP**

**LD SP,HL**

**PUSH IY**

Он выделит 10 байт стекового пространства для локальных переменных. Кадр стека после выполнения этого кода будет выглядеть так, как показано на [Рисунке 5-1](#r51).

Рисунок 5‑1 Кадр стека после входа в функцию

**│ │ │ Стек растет вниз, в сторону**

**│ Аргументы │ │ меньших адресов памяти**

**│ │ ▼**

**├──────────────────────┤**

**│ Адрес возврата │**

**├──────────────────────┤**

**│ Сохраненный IX │ IX ссылается сюда**

**├──────────────────────┤◀──**

**│ │**

**│ Локальные переменные │**

**│ │**

**├──────────────────────┤**

**│ Сохраненный IY │ SP ссылается сюда**

**└──────────────────────┘◀──**

Все ссылки на локальные данные или параметры осуществляются через IX. Первый аргумент находится в (IX+4). Обратите внимание, что если IX не используется (то есть отсутствуют аргументы, использующие стек или локальные переменные, доступные в функции), то IX не будет сохранен. Аналогично, IY будет сохранен, только если он используется в качестве регистровой переменной.

Параметр может занять любое число байтов, в зависимости от его типа. Если символ передается как аргумент функции без прототипа (т.е. в стиле K&R), он расширяется до длины int. Если функция имеет прототип, аргумент char будет занимать только один байт, но для него в стеке будут выделены два байта.

Локальные переменные доступны в расположениях (IX-1) в сторону уменьшения. IY может использоваться в качестве регистровой переменной. Если используется глобальная оптимизация, другие регистры также могут использоваться в качестве регистровых переменных, но не сохраняются, поскольку они рассматриваются в качестве временных регистров при вызове функций.

Выход из функции осуществляется, инвертировав код входа. Он восстанавливает старые значения IX и IY, сбрасывает указатель вершины стека, чтобы указать на адрес возврата, извлекает адрес возврата в регистр, удаляет из стека аргументы и осуществляет переход к сохраненному адресу возврата. Пример кода завершения для функции с аргументами из 4 байтов в стеке следующий:

**POP IY**

**LD SP,IX**

**POP IX**

**POP BC**

**POP AF**

**POP AF**

**PUSH BC**

**RET**

Приведенный выше код временно сохраняет адрес возврата в регистре BC, удаляет 4 байта параметров в стеке, выталкивая их в AF, затем возвращает, продвигая обратный адрес возврата в стек и выполняет инструкцию RET. Обратите внимание, что функции без прототипов и функции с переменными списками аргументов, оставляют удаление аргументов из стека функции вызова.

## Передача аргумента функции

Некоторые аргументы функции передаются в регистрах DE и BC. Это происходит, если рассматриваемая функция имеет прототип в стиле ANSI и один из первых двух аргументов - слово или байт, а второй аргумент не является символом с многоточием (...). В этом случае первый аргумент будет помещен в DE (для слова) или E (для байта), а второй аргумент - в BC или C.

В зависимости от функции параметры на основе регистров могут или не могут храниться в стеке на некотором этапе во время выполнения функции. Эта особенность не имеет ничего общего с объявлением параметра register. Любые аргументы, передаваемые в стеке, будут сдвигаться строго справа налево. Самый левый аргумент будет занимать самый младший адрес в стеке.

Например, возьмем следующую функцию в ANSI-стиле:

**void**

**test(int a, int b, int c)**

**{**

**}**

Функция test() получит параметр **a** в регистре DE, параметр **b** в регистре BC и параметр **c** в стеке. Вызов test(1,2,3) сгенерирует следующий код:

**LD HL,3**

**PUSH HL**

**LD DE,1**

**LD BC,2**

**CALL \_test**

Обратите внимание, что test удаляет свои собственные параметры из стека, таким образом вызывающей стороне не требуется удалять параметр **c** из стека.

Для функций, объявленных с прототипом в стиле ANSI, ответственность за удаление параметров из стека лежит на вызываемой функции, а не на вызывающей. Например, функция с 10 байтами параметров в стеке вышла бы с кодом, подобным следующему:

**POP BC ; BC = адрес возврата**

**LD HL,10**

**ADD HL,SP ;** **Коррекция SP**

**LD SP,HL ; на 10 байтов**

**PUSH BC ; Адрес возврата -> стек**

**RET**

Функции, объявленные со списками аргументов в стиле K&R, не используют параметры register и оставляют не обработанный стек вызывающей стороне. Функции возвращающие значения long или float и использующие больше чем 16 байтов стека для аргументов, также оставят стек не обработанный вызывающей стороне. Функция стиля K&R (то есть функция с аргументами C старого стиля) очистит все локальные переменные из стека и завершит работу с помощью инструкции RET, предоставляя вызывающей стороне право очистить параметры из стека. Например, возьмем функцию старого стиля:

**void test(a,b,c)**

**int a,b,c;**

**{**

**}**

Функция test() получает параметры **a**, **b** и **c** в стеке, помещенными по порядку слева направо, начиная с самого нижнего адреса. Параметры стека удаляются из стека вызывающей функцией. Например, вызов test(1,2,3) сгенерирует следующий код:

**LD HL,3**

**PUSH HL**

**LD HL,2**

**PUSH HL**

**LD HL,1**

**PUSH HL**

**CALL \_test**

**POP BC**

**POP BC**

**POP BC**

Эти правила следует учитывать при написании процедуры на языке ассемблера, вызываемые из кода C. Часто полезно написать фиктивную функцию C с типами параметров аналогичными функции на ассемблере и скомпилировать в ассемблерный код с помощью опции ZC -S, позволяющей исследовать сгенерированный код входа и выхода. Таким же образом полезно изучить код, сгенерированный вызовом функции с тем же списком аргументов, как у функции на ассемблере.

## Значения возвращаемые функцией

Значения возвращаемые функцией передаются в вызывающую функцию следующим образом:

### Возвращение 8 разрядных значений

Восьмиразрядные значения (char, unsigned char) возвращаются в регистре L. Возвращаемые 8-ми разрядные значения не расширяются в регистровую пару HL. Например, в операторе:

**ch = char\_func()**

функция **char\_func()**, возвращающая символ, сгенерирует следующий код:

**CALL \_char\_func**

**LD A,L**

**LD (\_ch),A**

### Возвращение 16 разрядных значений

16-разрядные значения (int, unsigned int, short, unsigned short и pointer) возвращаются в паре регистров HL с младшим значащим байтом в регистре L. Например, в операторе:

**i = int\_func()**

функция **int\_func()**, возвращающая целое значение, сгенерирует следующий код:

**CALL \_int\_func**

**LD (\_i),HL**

### Возвращение 32 разрядных значений

32-разрядные значения (long, unsigned long, float и double) возвращаются в регистровых парах DE и HL. Наименее значимое слово находится в регистровой паре DE. В операторе:

**l = long\_func()**

функция **long\_func()**, возвращающая 32 разрядное значение, сгенерирует код:

**CALL \_long\_func**

**LD (\_l),DE**

**LD (\_l+2),HL**

### Возвращение значений структуры

Составные возвращаемые значения (struct и union) возвращаются путем копирования возвращаемого значения в статический буфер в psect bss и возвращения указателя на буфер в регистровой паре HL. Указатель структуры, возвращенный в HL, затем используется либо для ссылки на члены структуры, либо для копирования возвращаемого значения в *левостороннее выражение оператора* (lvalue). Рассмотрим следующий код на языке C:

**struct tst {**

**int i, j, k;**

**}**

**extern struct tst test(void);**

**int i;**

**struct tst s;**

**main()**

**{**

**s = test();**

**i = test().j;**

**}**

Компилятор сгенерирует следующий код для этих двух присвоений.

**;z.c: 12: s = test();**

**CALL \_test**

**LD DE,\_s**

**LD BC,6**

**LDIR**

**;z.c: 13: i = test().j;**

**CALL \_test**

**INC HL**

**INC HL**

**LD C,(HL)**

**INC HL**

**LD B,(HL)**

**LD (\_i),BC**

В первом случае вся структура присваивается другой переменной, поэтому указатель возврата в HL используется как указатель источника для перемещения блока в lvalue. Во втором случае ссылается только один элемент возвращаемого значения, поэтому указатель возврата используется для доступа к полю .j без создания локальной копии структуры.

При использовании возвращаемых значений структуры необходимо учитывать дополнительное пространство psect bss, выделенное для "скрытой" копии возвращаемого значения. В приведенном выше случае код, сгенерированный для функции test(), будет содержать объявление 6-байтного буфера, используемого для настройки возвращаемого значения.

## Соглашения о вызовах функций в большой модели

При использовании большой (с переключением банков) модели соглашения о вызовах подобны малой модели, за исключением фактического вызова. Вместо того, чтобы непосредственно вызвать функцию, в регистр А загружается номер банка из которого должна быть вызвана функция, в HL загружается ее адрес в этом банке, а затем выполняется вызов подпрограммы в общей памяти, которая выполняет необходимое переключение банка прежде чем перейти к функции. Текущий банк сохраняется в стеке.

При возврате из функции старый номер банка извлекается из стека перед возвратом в вызывающую функцию. Код на Z80, выполняющий это показан ниже. Обратите внимание, что существует две процедуры для возврата. Одна из них удаляет аргументы из стека (**lretp**), а другая - нет (**lret**).

Когда функция вызывается подобным образом, компилятор учитывает дополнительное слово в стеке при вычислении смещений аргументов.

**globallcall, lret, lretp, \_\_Lbasecode**

**psect lowtext,class=CODE**

**BBR equ 39h ;** **Базовый регистр банка**

**; lcall - выполнить дальний вызов**

**lcall:ex af,af' ; Сохранить номер нового банка**

**in0 a,(BBR) ; Получитб текущий банк**

**push af ; Сохранить его**

**ex af,af' ; Восстановить тщвый банк**

**sub \_\_Lbasecode/1000h ;** **Скорректировать номер банка**

**out0 (BBR),a**

**jp (HL) ; Перейти н новой процедуре**

**lret: pop af ; Восстановить номер банка**

**out0 (BBR),a ; Выбрать его**

**ret**

**lretp:push bc ; Получить скорректированный**

**exx ;** **Получить некоторые неиспользованные регистры**

**pop hl ; Скорректированное значение**

**pop af ; Номер банка**

**pop de ; Адрес возврата**

**add hl,sp**

**ld sp,hl ; Удалить параметры**

**push de ; Поместить адрес возврата назад**

**exx**

**out0 (BBR),a ; Восстановить банк**

**ret ; Готово**

### Функции near и basenear в большой модели

При использовании большой (с переключением банков) модели функции по умолчанию вызываются используя описанный выше механизм. Однако можно определить функции, вызываемые с помощью простой последовательности call/ret, тем самым ускоряя код. Это можно сделать двумя способами: с помощью функций near и basenear.

Функция с атрибутом near может вызываться только из одного и того же банка, в то время как функция basenear находится в общей области 0 и может быть вызвана из любого банка. Функции near должны быть объявлены static и вызываться только из того же модуля. Следующий код показывает пример функции такого рода.

**static near int**

**read\_port( void)**

**{**

**while(STATUS & 0x80)**

**;**

**return DATA;**

**}**

**basenear void**

**kick\_dog( void)**

**{**

**WATCHDOG = 1;**

**}**

## Стек и выделение кучи

Как упоминалось ранее, стек растет вниз. При запуске указатель стека инициализируется в верхней части доступной памяти. Для CP/M это начало BDOS. Для встроенного кода он определяется адресом и размером ОЗУ, определенными во время компоновки. Стек устанавливается как сумма этих двух значений.

Куча (heap) - динамично выделяемая область памяти, с помощью sbrk(), callok() или malloc(). Она растет вверх от вершины статически выделенной памяти. Она определяется вершиной psect bss, которой компоновщик присваивает имя \_\_Hbss. Функция sbrk() проверяет объем памяти, находящийся между вершиной кучи и нижней частью стека, и если после предоставления по запросу sbrk() остается меньше, чем 1 Кбайт, отклоняет его. Это значение может быть изменено, редактированием sbrk.as, повторным ассемблированием и заменой его в соответствующей библиотеке (например, Z80-SC.lib).

## Локальные переменные

Язык C поддерживает два класса локальных переменных в функциях: *автоматические* (auto) переменные, которые обычно выделяются в каком-либо стеке и *статические* переменные, которым всегда назначаются фиксированные ячейки памяти.

### Переменные auto

По умолчанию локальные переменные имеют тип auto. Локальная переменная имеет тип auto, если явно не объявлена как static. Переменные auto выделяются в стеке и ссылаются путем индексации регистра IX. Переменные не обязательно будут выделяться в объявленном порядке, в отличие от параметров, которые всегда находятся в лексическом порядке. Обратите внимание, что большинство квалификаторов типа не может использоваться с автоматическими переменными, так как отсутствует управление местом хранения. Исключениями являются const и volatile.

### Статические переменные

Статическим (static) переменным выделяются память в psect bss и они занимают фиксированные ячейки памяти, которые не перекрываются с областью памяти других функций. Статические переменные локальны по отношению к функции, в которой они объявлены, но могут быть доступны из другой функции с помощью указателей. Статические переменные гарантированно сохраняют свое значение между вызовами функции, если явно не изменены через указатель. Статические переменные не имеют ограничений, связанных с архитектурой Z80.

## Секции psect генерируемые компилятором

Компилятор разбивает код и объекты данных на несколько стандартных секций программы (называемых "psect"). Ассемблер HI-TECH позволяет включать в ассемблерный код произвольное число именованных psect. Компоновщик сгруппирует все данные конкретной psect в один сегмент. Если вы используете ZC или HPDZ для вызова компоновщика, то не должны волноваться об информации, приведенной здесь, за исключением получения дополнительных знаний. Если вы хотите выполнить компоновщик вручную, или написать свои собственные подпрограммы на ассемблере, необходимо внимательно прочитать этот раздел. При генерации кода компилятор использует следующие psect:

**vectors** Секция psect vectors, содержит вектор сброса и вектор NMI. Секция vectors при компоновке обычно размещается в ПЗУ по адресу 0 с тем, чтобы Z80 выбрал содержание вектора сброса по нулевому адресу при аппаратном сбросе. Код в эту psect привносится из модуля времени выполнения startoff.

**lowtext** используется в большой модели для кода, который должен войти в нулевую общую область, например, код запуска.

**text** используется для всего исполняемого кода в малой модели. Написанные пользователем подпрограммы на ассемблере должны также быть помещены в psect text. При использовании большой модели, код C помещается в секции psect ltext, который будет распределен по различным банкам памяти, но некоторые библиотечные функции помещаются в psect text в общей области. Все функции обработки прерываний помещаются в psect text.

Если используются несколько ПЗУ (только в малой модели) то, содержание psect text может быть распределено по нескольким диапазонам памяти. Код в эту psect поступает из модулей и библиотек пользователя.

**ltext** В большой (с переключением банков) модели код программы из пользовательских модулей (и модулей библиотек, написанных на языке C), помещается в локальные секции psect, с именем ltext, а не в psect text. Это позволяет компоновщику присваивать эти psect банкам. Каждая psect ltext будет иметь различные адреса загрузки и ссылки. Адрес загрузки - физический адрес в ПЗУ, адрес ссылки будет в переключаемой области в 64K логическом адресном пространстве. Так как это локальные psect, на них нельзя ссылаться по имени в командной строке компоновщика, поэтому определен класс psect basecode, содержащий эти psect. Этот класс используется в командной строке компоновщика, чтобы обобщенно ссылаться ко всем секциям psect ltext.

**strings** Секция psect strings используется для всех неименованных строковых констант, таких как строковые константы, передаваемые в качестве аргументов подпрограммы, таких как printf() и puts(). Эта psect соединяется в ПЗУ, поскольку она не нуждается в модификации.

**const** используется для всех инициализированных констант класса const. Например:

**const char masks[] = { 1,2,4,8,16,32,64,128 };**

Эта psect компонуется в ПЗУ, так как она не должна изменяться.

**im2vecs** При использовании прерываний в режиме 2 необходимо иметь блок памяти, выделенный для таблицы векторов прерываний второго режима. Эта таблица должна начинаться на границе 256 байт и может быть длиной до 256 байтов. Эта psect представляет данный блок памяти. Она будет скомпонована в ПЗУ после всех других секций psect в ПЗУ и выровнена на 256-байтовой границе.

**basecode** Секция psect нулевой длины, определяющая физический адрес переключаемого ПЗУ. В сочетании с размером области переключаемого банка она используется для генерации адресов кода с переключением банков и номера банка, используемого во время выполнения.

**baseram** другая секция psect нулевой длины, используемая для определения физического адреса ОЗУ. Она позволяет коду startoff времени выполнения Z180 устанавливать отображение общей области 1 в ОЗУ.

**ramstart** еще одна psect нулевой длины, предназначенная для указания начала области ОЗУ (общая область 1 в модели Z180 с переключением банков). В зависимости от других опций компоновщика psect data или bss будет расположена в начале ОЗУ.

**data** Секция psect data содержит все статически инициализируемые данные, за исключением относящихся к классам code или const.

Секция psect data соединяются в ПЗУ, поскольку данные инициализируются и копируются в ОЗУ при запуске, если в опциях во время компоновки не указано иное. Если данные psect data останутся в ПЗУ, ОЗУ не будет занята ими, но они не могут быть изменены во времени выполнения.

**bss** Секция psect bss используется для всех неинициализированных статических (static) и внешних (extern) переменных, находящихся в ОЗУ. Вся область секции bss заполняется нулями при выполнении кода startoff перед вызовом main().

**stack** Секция используемый для определения вершины ОЗУ. Он не содержит данных и всегда имеет нулевую длину. Начальный адрес этой psect (из ассемблерного значения \_\_Lstack) загружен в указатель стека во времени выполнения кода startoff.

**heap** еще одна psect нулевой длины, используемая для определения верхней границы задействованного ОЗУ. "Куча" (heap) во время выполнения, используется для выделения динамической памяти, начнется с этого места и растет в сторону стека (который растет вниз).

**nvram** Эта psect используется для хранения сохраняемых (persistent) переменных. Ей может быть присвоен абсолютный адрес во время компоновки, или по умолчанию драйвер компилятора свяжет ее с секцией psect bss. Она не очищается и не изменяется при запуске.

## Модули начальной загрузки времени выполнения

Начальный адрес программы C обычно является наименьшим адресом кода. Для приложений ПЗУ это самый нижний адрес ПЗУ, обычно 0h. Для CP/M начальный адрес - начало TPA, 100h. Глобальный символ 100h находится по этому адресу. Код, расположенный по начальному адресу, выполняет некоторую инициализацию, в частности, очистку psect bss (неинициализированные данные) и инициализацию указателя вершины стека. В случае CP/M он также вызывает подпрограмму, установки значений argv и argc, которые передаются в main(). В зависимости от опций времени компиляции эта подпрограмма может разворачивать или не разворачивать подстановочные символы в именах файлов (? и \*), и выполняет перенаправление ввода-вывода.

При создании списка аргументов, загрузочный код вызывает функцию \_main. Обратите внимание на символ подчеркивания "\_" перед именем функции: все символы, полученные из внешних имен в программе C, содержат этот символ, добавленный компилятором. Это помогает предотвратить конфликты с ассемблерными именами. По определению языка C функция main() является "основной программой".

Код запуска времени выполнения обеспечивается стандартным модулем, находящимся в каталоге LIB, выбранном из перечисленных в [Таблице 5-6](#t56) на странице 117.

Таблица 5‑6 Стандартные модули startoff времени выполнения

|  |  |  |
| --- | --- | --- |
| **Модель** | **Модуль startoff Z80** | **Модуль startoff Z180** |
| **Малая** | RTZ80-S.OBJ | RTZ180S.OBJ |
| **CP/M** | RTZ80-C.OBJ | RTZ180C.OBJ |
| **Большая** | RTZ80-L.OBJ | RTZ180L.OBJ |

Смотрите ассемблерный файл соответствующий объектному файлу для ознакомления с исходным кодом модуля запуска.

### Процедура powerup

Часто некоторые аппаратные конфигурации требуют особой инициализации, в течение первых нескольких циклах выполнения после сброса. Вместо того, чтобы изменять модуль запуска (startoff) времени выполнения, для достижения этого имеется ловушка вектора сброса, предоставляемая через подпрограмму powerup. Это предоставленный пользователями ассемблерный модуль, который при сбросе будет немедленно выполнен. Часто он может быть встроен в модуль C в качестве встроенного ассемблерного кода.

Процедура powerup вызывается с указателем стека, который указывает на поддельный обратный адрес, встроенный в код времени выполнения startoff, таким образом, для работы стека не требуется ОЗУ. Программа powerup должна быть написана, в предположении, что ОЗУ мало или отсутствует, и использовать системные ресурсы только после их тестирования и включения. Следующий пример кода, встроенный в файл C, ожидает стабилизации динамического ОЗУ после сброса, а затем выполняет проверку ОЗУ. Если тест пройден, он возвращается (через жестко закодированный обратный адрес), в противном случае он зажигает светодиод и зависает:

**#asm**

**;**

**PSECT text**

**GLOBAL powerup**

**; Пример подпрограммы powerup - проверка ОЗУ, оставляет**

**; включенным светодиод и зависает при неисправности ОЗУ**

**;**

**; Предполагает ОЗУ: с 2000H до 3FFFH, LED: порт 0FEH**

**;**

**LED EQU 0FEH**

**RAMSTART EQU 2000H**

**RAMSIZE EQU 2000H**

**;**

**; Цикл, для стабилизации DRAM**

**;**

**tbytes: DEFB 55H,0AAH,0FFH,0**

**;**

**powerup: LD BC,0FFFFH**

**1: DEC BC**

**LD A,B**

**OR C**

**JR NZ,1b**

**;**

**; Основной цикл проверки - проверка ОЗУ с каждым из 4**

**; тестовых значений: 55H,0AAH,0FFH и 0**

**;**

**LD B,4**

**LD HL,tbytes**

**2: LD A,1**

**OUT (LED),A ; Выключить светодиод**

**LD A,(HL) ; Загрузить тестовое значение**

**EXX ; Сохранить HL & B**

**LD HL,RAMSTART**

**LD DE,RAMSTART+1**

**LD BC,RAMSIZE-1**

**LD (HL),A**

**EX AF,AF' ; Сохранить A**

**LDIR ; Заполнение памяти**

**;**

**; Цикл на некоторое время, оставляя время для**

**; отображения проблем с обновлением**

**;**

**LD BC,0FFFFH**

**3: DEC BC**

**LD A,B**

**OR C**

**JR NZ,3b**

**;**

**LD A,0**

**OUT (LED),A ; Выключить светодиод**

**EX AF,AF' ; Восстановить A**

**;**

**; Цикл тестирование памяти с сохраненным значением**

**;**

**LD HL,RAMSTART**

**LD BC,RAMSIZE**

**4: CP (HL) ;** **Проверить на соответствие**

**JR NZ,ramerr**

**INC HL**

**DEC BC**

**LD A,B**

**OR C**

**JR NZ,4b**

**;**

**EXX ; Восстановить регистры**

**INC HL ; Следующее тестовое значение**

**DJNZ 2b**

**RET**

**;**

**; ramerr: Включить светодиод и зависнуть**

**;**

**ramerr: LD A,1**

**OUT (LED),A**

**hang: JR hang**

**;**

**#endasm**

По умолчанию стандартные библиотеки содержат подпрограммы powerup, показанные в [Таблице 5-7](#t57) на странице 120.

Таблица 5‑7 Действия powerup по умолчанию

|  |  |  |
| --- | --- | --- |
| **Модель/Процессор** | **Библиотека** | **Действия powerup по умолчанию** |
| **Малая/Z80** | Z80-SC.lib | Нет |
| **Большая/Z80** | Z80-LC.lib | Нет дествий по умолчанию: необходимо обеспечить свою собственную подпрограмму |
| **Малая/Z180** | Z801SC.lib | Устанавливает общую область 1 для ссылки на ОЗУ |
| **Большая/Z180** | Z801LC.lib | Устанавливает область банка и общую зону 1 |

### Использование символов определенных компоновщиком

Чтобы код startoff времени выполнения мог очистить psect bss и скопировать данные psect, он должен определить адрес загрузки, адрес ссылки и размер этих psect. Это достигается с помощью специальных символов, созданных компоновщиком.

Адрес ссылки psect может быть получен из значения глобального символа с именем \_\_Lname, где name - имя psect. Например, \_\_Lbss - нижняя граница psect bss. Самым высоким адресом psect (т.е. адрес ссылки плюс размер) является символ \_\_Hname. Если psect имеет различные адреса загрузки и ссылки, как может иметь место, если данные psect соединены для выполнения в ОЗУ, адрес загрузки определяется \_\_Bname.

#### Очистка psect bss

В стандартной модели времени выполнения Z80 psect bss имеет одинаковые адреса ссылки и загрузки, таким образом, \_\_Bbss не определен. Секция bss очищается, обнулением \_\_Hbss-\_\_Lbss байтов памяти, начиная с адреса \_\_Lbss.

#### Копирование psect data

Данные psect соединяются в ОЗУ и ПЗУ. Код времени выполнения startoff должен скопировать образ ПЗУ в ОЗУ. Это достигается, копированием \_\_Hdata-\_\_Ldata байтов начиная с адреса \_\_Bdata в ПЗУ в \_\_Ldata в ОЗУ.

#### Инициализация стека

Когда ZC или HPDZ вызывают компоновщик, они включают psect, названную stack в опциях компоновщика. Секция psect stack не содержит объекты данных и используется только в качестве удобного метода определения начального значения указателя вершины стека. Адрес, заданный в ZC или HPDZ секции psect stack, является определенным пользователем адресом ОЗУ плюс определенным пользователем размером ОЗУ. Например, если используется опция ZC -A0,8000,4000, то psect stack получит адрес 8000H + 4000h = 0C000h. Код времени выполнения startoff загружает значение \_\_Lstack в указатель вершины стека до выполнения любого кода пользователя.

### Настройка кода Startoff времени выполнения

Если обнаружится, что заканчивается место в ПЗУ, можно уменьшить размер ПЗУ, настраивая код времени выполнения startoff. Стандартный модуль времени выполнения startoff содержит код очистки psect bss и копирования psect data. Размер кода должен быть сокращен путем создания пользовательской версии кода startoff, который выполняет только те инициализации, которые требуются вашему приложению. Исходный код времени выполнения startoff может быть найден в подкаталоге **SOURCES**, если он установлен. Представлены несколько различных версий модулей времени выполнения startoff, перечисленные в [Таблице 5-6](#t56) на странице 117.

#### Сообщение авторского права

Код времени выполнения startoff включает в себя сообщение об авторском праве HI-TECH C Software, которое размещается компоновщиком в начале ПЗУ. Если свободного места в ПЗУ недостаточно, можно удалить сообщение об авторском праве, немного освободив ПЗУ. Если места в ПЗУ достаточно, сообщение об авторском праве HI-TECH Software должно остаться неизменным. Также можно добавить свое собственное сообщение об авторском праве.

#### Использование нового кода времени выполнения startoff

После изменения кода времени выполнения startoff в соответствии со своими потребностями повторно выполните ассемблирование с помощью команды:

**ZAS -x RTZ80xx.AS**

где RTZ80xx.AS - имя измененного модуля startoff. Код времени выполнения startoff должен быть скопирован в подкаталог библиотеки. Для большинства инсталляций это будет C:\HT-Z80\LIB. После того, как измененный код startoff появится в каталоге библиотеки, нужно только повторно скомпоновать свое приложение для использования ваших изменений. При использовании адаптированного кода времени выполнения startoff, нужно быть осторожным при внесении дополнительных изменений в приложение. Если задействован минимальный модуль startoff и добавляется код, использующий psect bss, или данные расположенные в ОЗУ, могут возникнуть проблемы. Без кода очистки bss любые неинициализированные переменные во внешнем ОЗУ не будут обнулены до вызова main(). Что еще более важно, если используются инициализируемые данные, но отсутствует код копирования, при запуске статически инициализированным переменным не будут установлены их правильные значения.

## Оптимизация кода Z80

Необходимо избегать написания кода, который будет большим или неэффективным. Для увеличения скорости выполнения и уменьшения размера кода, можно использовать некоторые или все из этих предложений:

* Используйте 8-разрядные величины (char или unsigned char), где это необходимо, а не 16-разрядные величины, поскольку они занимают меньше места и доступ к ним может быть получен быстрее.

## Директивы pragma

Имеются предопределенные директивы компилятора, которые могут использоваться для изменения поведения компилятора. Они реализованы при помощи стандартного объекта ANSI **pragma**. Формат pragma следующий:

**#pragma ключевое\_слово опции**

где **ключевое\_слово** - одно из перечня ключевых слов, некоторые из которых сопровождаются определенными **опциями**. Список ключевых слов приведен в [Таблице 5-8](#t58) на странице 122. Каждое ключевое слово описано ниже.

Таблица 5‑8 Директивы pragma

|  |  |  |
| --- | --- | --- |
| **Директива** | **Назначение** | **Пример** |
| **jis** | Включает в строках обработку символа JIS[[16]](#footnote-16) | #pragma jis |
| **nojis** | Отключает в строках обработку символа JIS (по умолчанию) | #pragma nojis |
| **printf\_check** | Включает проверку формата строк printf-стиля | #pragma printf\_check(printf) |
| **psect** | Переименовывает psect, определенную компилятором | #pragma psect text=mytext |
| **strings** | Определяет квалификатор строковых констант | #pragma strings code |
| **switch[[17]](#footnote-17)** | Определяет генерацию кода для операторов switch | #pragma switch direct |

#### Директивы #pragma jis и nojis

Если код содержит строки с двухбайтовыми символами в кодировке JIS для японского языка и других национальных символов, директива **#pragma jis** обеспечивает правильное обращение с этими символами, в частности не интерпретируя символ наклонной черты влево (\), когда он появляется в качестве второй половины двухбайтового символа. Директива **nojis** отключает эту специальную обработку. Обработка символа JIS по умолчанию отключена.

#### Директива #pragma printf\_check

Некоторые функции библиотеки принимают строку формата, за которой следует переменное количество аргументов в виде printf(). Хотя строка формата интерпретируется во время выполнения, она может быть проверена во время компиляции на согласованность с остальными аргументами. Эта директива включает эту проверку для именованной функции, например, системный заголовочный файл <stdio.h> включает директиву **#pragma printf\_check (printf)**, чтобы включить эту проверку printf(). Вы можете также использовать ее для любой определяемой пользователем функции, которая принимает строки формата в стиле printf. Обратите внимание, что уровень предупреждений должен быть установлен равным -1 или ниже для этой опции, чтобы иметь эффект.

#### Директива #pragma psect

Как правило, объектный код, сгенерированный компилятором, разбивается на стандартные psect, как описано ранее. Это нормально для большинства приложений, но иногда необходимо перенаправить переменные или код в различные psect, когда нужна специальная конфигурация памяти. Например, если аппаратные средства включают область памяти с аварийным батарейным питанием, может быть желательно перенаправить некоторые переменные из bss в psect, которая не очищается при запуске (хотя эта функция предусмотрена в качестве стандартной функции). Код и данные для любой стандартной C psect могут быть перенаправлены, используя директиву **#pragma** **psect**. Например, если весь исполняемый код, сгенерированный определенным исходным файлом C, должен быть помещен в psect, названную altcode, должна использоваться следующая директива:

**#pragma psect text=altcode**

Эта директива сообщает компилятору, что все, что обычно помещается в psect text, теперь должно быть помещено в psect altcode. Любая заданная psect может быть перенаправлена только один раз в конкретном исходном файле, и все переадресации psect для определенного исходного файла должны быть размещены в верхней части файла под любыми #include и над любыми другими объявлениями. Например, чтобы объявить группу неинициализированных переменных, которые размещены в psect названной xram, следует использовать следующий метод:

**---Файл XRAM.C**

**#pragma psect bss=xram**

**char buffer[20];**

**int var1, var2, var3;**

Любые файлы, которым необходимо получить доступ к переменным, определенным в XRAM.C, должны включать следующий заголовочный файл:

**---Файл XRAM.H**

**extern char buffer[20];**

**extern int var1, var2, var3;**

Директива #pragma psect позволяет разбивать код и данные на произвольные области памяти. Определения кода или данных для нестандартных psect должны быть сохранены в отдельных исходных файлах, как описано выше. При связывании кода, который использует нестандартные имена psect, вы не сможете использовать опцию ZC -A для указания адресов ссылок для новых psect, вместо этого вам нужно будет использовать опцию ZC -L, чтобы указать дополнительную опцию компоновщика, используя компоновщик вручную или используя проект HPDZ для компиляции и привязки кода. Если вам нужна стандартная конфигурация с добавлением только дополнительной psect, такой как xram, вы можете использовать опцию ZC -L, чтобы добавить дополнительную спецификацию -P к команде компоновщика. Например:

**ZC -L-Pxram=1000h/20000h -A0,8000,8000 test.obj xv.obj**

соединит TEST.OBJ и NV.OBJ со стандартной конфигурацией ПЗУ в 0h, ОЗУ в 8000h, и дополнительная psect xram в 1000h в ОЗУ, но не перекрывая любой допустимый адрес загрузки ПЗУ. При использовании интегрированной среды HPDZ, можно настроить файл проекта при выборе **Start** **New** **Project** (Начать новый проект), добавить имена четырех исходных файлов, используя **Source Files** **...** (Исходные файлы...) и затем изменить опции компоновщика, включая любой новый psect при выборе **Linker Options...** (Опции компоновщика...).

#### Директива #pragma strings

Любые определяемые пользователем переменные могут уточняться многими квалификаторами типа (см. [Раздел 5.9](#_Специальные_квалификаторы_типа) "Специальные квалификаторы типа" на странице 96), но константные строки (т.е. анонимные строки, встроенные в выражения) обычно являются неквалифицированными. Это означает, что они будут помещены в сегмент данных. Чтобы управлять этим поведением, директива **#pragma psect strings** позволяет вам указать набор квалификаторов, которые будут применяться ко всем последующим константным строкам. Если квалификатор будет определен, он будет добавлен к любым квалификаторам, определенным ранее. Использование директивы без квалификатора удалит все квалификаторы из любых последующих строк, то есть восстановит нормальное состояние.

Например, чтобы квалифицировать стоки с *кодом*, следует использовать пример, приведенный в [Таблице 5-8](#t58) на странице 122. Обратите внимание, что все константные строки будут тогда иметь *указатель на код типа char*, и не будут применимы, где ожидается простой *указатель на char*.

#### Директива #pragma switch

Компилятор генерирует несколько различных видов кода для операторов switch. Обычно компилятор выбирает самый маленький код для данного switch. Основными методами являются direct (прямой) switch, при котором таблица переходов индексируется значением включения, и simple (простой) switch, использующий последовательность сравнений. direct switch выполняется за постоянное время и, как правило, будет быстрее, чем simple switch, но может быть достаточно большим для разреженного набора меток case. Если существует потребность в детерминированном по времени switch, с помощью этой директивы можно выбрать direct (прямой) switch. Директива **#pragma switch** оказывает действие только на непосредственно следующий оператор switch, и только если он появляется в той же функции. Возможными параметрами этой директивы являются auto, восстанавливающий поведение по умолчанию, direct, который выбирает прямой переключатель, и simple, который выбирает простой переключатель. Смотрите пример в [Таблице 5-8](#t58) на странице 122.

## Стандартные функции ввода-вывода и последовательный ввод-вывод

В библиотеке C с компилятором ZC предоставляется много стандартных функций ввода-вывода, в частности эти функции предназначены для чтения и записи форматированного текста в стандартный поток вывода и ввода. Список доступных функций приведен в [Таблице 5-9](#t59) на странице 124. Более подробно эти функции описаны в [Главе 10](#_Библиотечные_функции_1) "Библиотечные функции". Необходимо связать либо sersio.c (для Z80 SIO), или ser180.c (для Z180) прежде, чем любые символы, записанные или прочитанные с использованием этих функций, будут отправлены или получены. Посмотрите на эти файлы, расположенные в каталоге sources, чтобы убедиться в том, что они поддерживают ваше оборудование.

Таблица 5‑9 Поддерживаемые функции STDIO

|  |  |
| --- | --- |
| **Имя функции** | **Назначение** |
| **puts(char \* s)** | Пишет строку в stdout, добавляет новую строку |
| **char \* gets(char \* buf)** | Получает строку текста из stdin в буфер, удаляет новую строку |
| **printf(char \* s, ...)** | Отформатированная печать в stdout |
| **putchar(int c)** | Помещает один символ в stdout |
| **scanf(char \*, ...)** | Читает отформатированный ввода из stdin |
| **sprintf(char \* buf, char \* s, ...)** | Записывает форматированный текст в буфер |
| **sscanf(char \* buf, char \* s, ...)** | Читает отформатированный текст из буфера |
| **vprintf(char \* s, va\_arg list)** | Версия printf, принимающая список аргументов |
| **vscanf(char \* s, va\_arg list)** | Версия scanf, принимающая список аргументов |
| **vsprintf(char \* buf, char \* s, va\_arg list)** | Версия sprintf, принимающая список аргументов |
| **vsscanf(char \* buf, char \* s, ...)** | Версия sscanf, принимающая список аргументов |

Если код выполняется под отладчиком Lucifer, канал в ser180.c. Часть отладчика, относящаяся к хосту будет действовать как простой терминал. Если вы хотите отправлять или получать символы с помощью других средств (например, на ЖК-дисплей), вы должны изменить тот или иной модуль в sersio.c или ser180.c.

Чтобы заменить один из этих модулей на собственную версию, скопируйте исходный код из каталога SOURCES в файл в ваш рабочий каталог, внесите необходимые изменения и включите этот файл в свой проект - либо в список исходных файлов в HPDZ, либо в командной строке для ZC. Вы должны сохранить все функции, присутствующие в модуле, даже если некоторые из них не используются (в этом случае они могут быть пустыми функциями). Несоблюдение этого требования может привести к повторению определенных символьных сообщений во время компоновки.

# Макроассемблер Z80

Макроассемблер Z80 HI-TECH Software ассемблирует исходные файлы для микропроцессоров семейства Z80 и Z180. В этой главе описывается использование ассемблера и директив (псевдоопераций ассемблера и средств условной компиляции), распознаваемых ассемблером.

Пакет ассемблера HI-TECH включает компоновщик, библиотекарь, генератор перекрестных ссылок и конвертор объектного кода.

## Использование ассемблера

Ассемблер называется ZAS и доступен для операционных систем MS-DOS и UNIX. Обратите внимание, что ассемблер при отсутствии ошибок или предупреждений не производит сообщений и не выдает сообщение "Ассемблирование завершено".

Использование ассемблера в этих операционных системах схоже. Все параметры командной строки распознаются в верхнем или нижнем регистре. Основной формат команды следующий:

**zas [ опции ] файлы ...**

**Файлы** представляют собой список из одного или нескольких исходных файлов на языке ассемблере, разделенных пробелами. Если указано более одного исходного файла, ассемблер рассматривает их как один модуль, то есть будет выполнено одно ассемблирование с конкатенацией всех указанных исходных файлов. Файлы должны быть указаны полностью, не предполагается никаких расширений или суффиксов по умолчанию.

**Опции** - это необязательный список опций ассемблера, разделенных пробелами, каждый со знаком минус (-) в качестве первого символа. Полный перечень возможных опций приводится в [Таблице 6-1](#t61) на странице 126, далее приводится полное описание каждой опции.

Таблица 6‑1 Опции ассемблера ZAS

|  |  |  |
| --- | --- | --- |
| **Опция** | **Значение** | **По умолчанию** |
| **-C** | Производит перекрестные ссылки | Нет перекрестных ссылок |
| **-Eformat** | Устанавливает формат ошибки |  |
| **-I** | Перечисляет макрорасширения | Не перечисляет макросы |
| **-J** | Оптимизирует переходы к ответвлениям | Не оптимизирует переходы |
| **-Llistfile** | Производит листинг | Нет листинга |
| **-N** | Игнорирует арифметическое переполнение |  |
| **-Ooutfile** | Определяет имя объекта | srcfile.OBJ |
| **-Plength** | Определяет длину формы листинга | 66 |
| **-S** | Не сообщает об ошибке размера |  |
| **-U** | Не сообщает о неопределенных символах |  |
| **-V** | Производит информацию о номере строки | Без номеров строк |
| **-Wwidth** | Указывает ширину страницы листинга | 80 |
| **-X** | Не помещает локальные символы в файл OBJ |  |

## Опции ассемблера

Опции командной строки, распознаваемые ZAS, следующие:

**-C** Когда использоваться эта опция будет произведен файл перекрестных ссылок. Этот файл, названный srcfile.crf, где srcfile - основная часть имени первого исходного файла, будет содержать необработанную информацию о перекрестных ссылках. Затем должна быть выполнена утилита перекрестных ссылок CREF, чтобы произвести отформатированный список перекрестных ссылок.

**-E** Формат сообщения об ошибке по умолчанию представлен в форме:

**имя\_файла:строка: сообщение**

где ошибка с **сообщением** произошла в **строке** в файле **имя\_файла**. Опция -E2 создаст менее читаемый формат, который используется HPD.

**-I** Эта опция заставляет выводить в листинг макрорасширения и не ассемблированные условные конструкции, которые в противном случае были бы отключены с помощью ассемблерного управления \*LIST OFF. Опция -L по-прежнему необходима для создания листинга.

**-J** Опция запросит ассемблер попытаться ассемблировать переходы и условные переходы в виде относительных ветвей, где это возможно. Будут оптимизированы только условные переходы с эквивалентными ответвлениями, и только переходы к ответвлениям, в которых цель находится в пределах ветви. Обратите внимание, что использование этого параметра замедляет ассемблирование, из-за необходимости ассемблера, выполнить дополнительный проход по входному коду.

**-Llistfile** Эта опция запрашивает генерацию листинга ассемблирования. Если определен listfile, то листинг будет записан в этот файл, иначе он будет записан в стандартный вывод.

**-N** Подавляет нормальную проверку арифметического переполнения.

При обработке переполнения ассемблер следует "*Руководству по языку ассемблера Z80*", и в некоторых случаях это может привести к ошибке, хотя выражение, по сути, вычисляет то, что предполагал пользователь. Опция -N может использоваться для переопределения проверки переполнения.

**-O** По умолчанию ассемблер определяет имя создаваемого объектного файла, удаляя любой суффикс или расширение (т.е. часть после последней точки) из первого имени исходного файла и добавляя .OBJ. Опция -O позволяет пользователю переопределять значение по умолчанию и явно указывать имя объектного файла.

**-Plength** По умолчанию длина страницы листинга составляет 66 строк (11 дюймов по 6 строк на дюйм). Опция -P позволяет указать другую длину страницы.

**-S** Если ячейка памяти с размером байта инициализируется значением, которое слишком велико, чтобы соответствовать 8 битам, то ассемблер генерирует сообщение "Size error" (Ошибка размера). Использование опции -S подавит сообщения этого типа.

**-U** Если опция -U не задана, неопределенные символы, встретившиеся во время ассемблирования, рассматриваются как внешние, однако выдается сообщение об ошибке для каждого неопределенного символа. Использование этой опции подавляет только сообщения об ошибках, она не изменяет сгенерированный код.

**-V** Эта опция включает в создаваемый объектный файл, информацию о номере строки ассемблера и имени файла для использования отладчиком. Обратите внимание, что номера строк будут соответствовать строкам ассемблерного кода, При ассемблировании файла, созданного компилятором, строки и директивы файла будут вставлены компилятором, поэтому эта опция не требуется.

**-Wwidth** Эта опция позволяет указать ширину бумаги файла листинга listfile в символах. Ширина должна быть десятичным числом, большим, чем 41. По умолчанию ширина составляет 80 символов.

**-X** Объектный файл, создаваемый ассемблером, содержит информацию о символах, включая локальные символы, т.е. символах, которые не являются общедоступными или внешними. Опция -X препятствует включению локальных символов в объектный файл, тем самым уменьшая размер файла.

## Язык ассемблера Z80

Ниже описан исходный язык, принятый макроассемблером Z80 HI-TECH Software. Все мнемоники кода операции и синтаксис операндов строго соответствует ассемблеру Z80.

### Набор символов

Используемый набор символов является стандартным ASCII с 7 битами. Регистр символов значим для идентификаторов, но не для кодов операций и зарезервированных слов. Символы табуляции рассматриваются как эквивалентные пробелам.

### Константы

#### Цифровые константы

Ассемблер выполняет всю арифметику как 32 бита со знаком. Ошибки будут возникать, если величина будет слишком большой, чтобы поместиться в ячейку памяти. По умолчанию основание всех чисел равняется 10. Другие основания систем счисления могут быть определены завершающим спецификатором основания, как показано в [Таблице 6-2](#t62) на странице 128.

Шестнадцатеричные числа должны начинаться с цифры (например, 0ffffh), чтобы отличаться от идентификаторов. Шестнадцатеричные константы принимаются в верхнем или нижнем регистре.

Таблица 6‑2 Числа и основания ZAS

|  |  |
| --- | --- |
| **Система счисления** | **Формат** |
| **Двоичная** | Цифры 0 и 1 с последующим символом B |
| **Восьмеричная** | Цифры от 0 до 7 с последующим символом O, Q, o или q |
| **Десятичная** | Цифры от 0 до 9 с последующим символом D или d, или без него |
| **Шестнадцатеричная** | Цифры от 0 до 9, от A до F с последующим символом H или h |

*Обратите внимание* , что после двоичной константы должен быть символ B в верхнем регистре, поскольку b в нижнем регистре используется для ссылок назад на временные (числовые) метки.

Вещественные числа в обычном формате допускаются только в директивах DEFF. Экспонента и мантисса вещественного числа должны быть десятичными. При использовании директивы DEFF вещественные числа сохраняются в 32 разрядном формате HI-TECH C. Вещественное число должно включать десятичную точку, а экспонента и знак не обязательны. Если экспонента присутствует, она должна следовать за мантиссой без промежуточных пробельных символов (пробелов и табуляций).

#### Символьные константы

*Символьная константа* - единственный символ, заключенный в одинарные кавычки ('). Много символьных констант могут использоваться только в качестве операнда псевдооперации DEFM.

#### Константы кодов операций

Любой код операции Z80 может использоваться в качестве константы в выражении. Значение кода операции в этом контексте будет байтом, к которому ассемблировался бы код операции обычным способом. Если код операции будет 2-байтовым кодом операции (с байтом префикса CB или ED), то будет использоваться только второй байт кода операции. Это особенно полезно при установке векторов перехода. Например:

**ld a,jp ;** **команда перехода**

**ld (0),a ; 0 служит переходом к теплой загрузке**

**ld hl,boot ; сделанной здесь**

**ld (1),hl**

### Разделители

Все числа и идентификаторы должны быть разграничены пробельными символами, не буквенно-цифровыми символами или символом конец строки.

### Специальные символы

Есть несколько символов, которые являются особенными в определенных контекстах. Внутри тела макроса символ @ используется для конкатенации токена. В списке аргументов макроса угловые скобки < и > используются, чтобы заключить параметры макро в кавычки.

### Идентификаторы

*Идентификаторы* представляют собой определяемые пользователем символы, представляющие ячейки памяти или числа. Символ может содержать любое число знаков, взятых из алфавитных букв, цифр и специальных символов - доллара ($), вопросительного знака (?) и подчеркивания (\_). Первый символ идентификатора не может быть цифрой. Регистр букв имеет значение, например, Fred не является тем же символом, что и fred. Ниже приведены примеры идентификаторов:

**An\_identifier**

**an\_identifier**

**an\_identifier1**

**$$$**

**?$\_12345**

#### Значимость идентификаторов

Пользователи других ассемблеров, которые пытаются реализовать формы ввода данных для идентификаторов, должны отметить, что этот ассемблер не придает никакого значения какому-либо символу и не налагает никаких ограничений или ожиданий на использование символа. Имена psect (секций программы) и обычные символы занимают отдельные перекрывающиеся пространства имен, но кроме этого, ассемблеру все равно, используется ли символ для представления байтов, слов или еще чего либо. Для определения адресов битов или любого другого типа данных не требуется специальный синтаксис, и компилятор ассемблера не выдает никаких предупреждений, если символ используется в нескольких контекстах. Синтаксис инструкций и режим адресации предоставляет ассемблеру всю необходимую информацию для генерации правильного кода.

#### Идентификаторы генерируемые ассемблером

Если в макро-блоке используется директива LOCAL, ассемблер будет генерировать уникальный символ для замены каждого указанного идентификатора в каждом расширении этого макроса. Эти уникальные символы будут иметь вид ??nnnn, где nnnn - четырехзначное число. Пользователь должен избегать указания символов с одинаковой формой.

#### Счетчик адреса

Текущее расположение в разделе активной программы доступно через символ $.

#### Символы регистров

Все регистры доступны при помощи своих стандартных имен, например, BC, de и т.д. Регистр имен регистров не является существенным. Невозможно приравнять символ к регистру.

#### Метки

*Метка* - имя в начале оператора, заканчивающееся двоеточием, которому присваивается значение, равное текущему смещению в текущей psect (секции программы). Метка может быть любым символом (включая числовые метки). Метка не совпадает с именем макроса, которое также появляется в начале строки в объявлении макроса, но за ним не следует двоеточие.

#### Временные метки

Ассемблер реализует систему временных меток (в отличие от локальных меток, используемых в макросах), которая освобождает программиста от создания новых меток в блоке кода. Временная метка определена как числовая последовательность и может быть ссылкой на такую же числовую последовательностью с суффиксом 'f' или 'b'. При использовании с суффиксом 'f', ссылка метки - первая найденная метка с тем же числом, начиная от текущего расположения, и с другой стороны 'b' заставит ассемблер смотреть назад при поиске метки.

Например:

**entry\_point: ;** **На эту ссылаются из далека**

**ld b,10**

**1: dec c**

**jr nz,2f ;** **если не 0, переход вперед к 2:**

**ld c,8**

**djnz 1b ;** **Уменьшение на 1 и переходим назад к 1:**

**jr 1f ;** **это не переход к той же**

**;** **самой метке djnz**

**2: call fred ;** **переход сюда от jr nz,2f**

**1: ret ;** **переход сюда от jr 1f**

Обратите внимание, что даже при том, что имеются две метки **1:**, неоднозначность не возникает, так как каждая упоминается уникально. **djnz 1b** ссылается на метку расположенную в исходном коде выше, в то время как младший **jr 1f** ссылается на метку расположенную ниже. Тем не менее, чтобы избежать путаницы, не рекомендуется дублировать числовые метки внутри процедуры.

### Строки

*Строка* представляет собой последовательность символов, не считая возврата каретки или новой строки, заключенные в соответствующие кавычки. Можно использовать одиночные (') или двойные (") кавычки, но открывающие и закрывающие кавычки должны быть одинаковыми. Строка, используемая в качестве операнда директивы DB, может иметь любую длину, но строка, используемая в качестве операнда инструкции, не должна превышать 1 или 2 символа, в зависимости от размера требуемого операнда.

### Выражения

*Выражения* состоят из чисел, символов, строк и операторов. Операторы могут быть унарными (один операнд, например .NOT.) или бинарными (два операнда, например, +). Операторы, допустимые в выражениях, перечислены в [Таблице 6-3](#t63) на странице 131. Используются обычные правила, определяющие синтаксис выражений. Операторы, начинающиеся с точки "." должны быть разделены пробелами, таким образом

**label .and. 1**

допустимо, а

**label.and.1**

нет.

Таблица 6‑3 Операторы

| **Оператор** | **Значение** |
| --- | --- |
| **&** | Поразрядная операция 'И' |
| **\*** | Умножение |
| **+** | Сложение |
| **-** | Вычитание |
| **/** | Деление |
| **<** | Меньше, чем с учетом знака |
| **=** | Равно |
| **>** | Больше, чем с учетом знака |
| **<=** | Меньше или равно с учетом знака |
| **>=** | Больше или равно с учетом знака |
| **<>** | Не равно с учетом знака |
| **\** | Побитовое дополнение |
| **^** или **|** | Поразрядное логическое 'ИЛИ' |
| **and** или **.and.** | Поразрядная операция 'И' |
| **eq** или **.eq.** | Тест равенства |
| **gt** или **.gt.** | Со знаком больше, чем |
| **high** или **.high.** | Старший байт операнда |
| **low** или **.low.** | Младший байт операнда |
| **lt** или **.lt.** | Со знаком меньше, чем |
| **mod** или **.mod.** | Деления по модулю |
| **not** или **.not.** | Побитовое дополнение |
| **or** или **.or.** | Поразрядное логическое 'ИЛИ' |
| **shl** или **.shl.** | Сдвиг влево |
| **shr** или **.shr.** | Сдвиг вправо |
| **utl** или **.ult.** | Без знака меньше, чем |
| **ugt** или **.ugt.** | Без знака больше чем |
| **xor** или **.xor.** | Исключающее 'ИЛИ' |
| **seg** | Адрес сегмента (номер банка) |

Перечисленные операторы могут свободно объединяться как в константные, так и в перемещаемые выражения. Компоновщик HI-TECH разрешает перемещение сложных выражений, поэтому результаты выражений, включающие перемещаемые идентификаторы, не могут быть разрешены до момента компоновки.

### Формат оператора

Допустимые форматы операторов приведены в [Таблице 6-4](#t64) на странице 132. Вторая форма является законной только с определенными директивами, такими как MACRO, SET и EQU. Поле метки необязательное и, если существует, должно содержать один идентификатор. Поле имени обязательно и должно также содержать один идентификатор. Обратите внимание, что метка, если она присутствует, содержит в конце двоеточие. Любая часть оператора может появиться в любой позиции или любом месте строки с оператором.

Таблица 6‑4 Формат операторов ZAS

|  |  |  |  |
| --- | --- | --- | --- |
| Метка: | Код операции | Операнды | ;Комментарий |
| Имя | Псевдооперация | Операнды | ;Комментарий |
| ;Только комментарий | | | |

### Секции программы

*Программные секции* или psect - это способ группировки частей программы, даже если исходный код не может быть физически смежным в исходном файле или даже если он располагается в нескольких исходных файлах. Если не определены как ABS (абсолютные), то psect перемещаемы.

Секция psect идентифицируется по имени и имеет несколько атрибутов. Директива psect используется для определения секций psect. В качестве аргументов требуется имя и необязательный список флагов, разделенных запятыми. Для получения полной информации смотрите [Раздел PSECT](#g63113) на странице 134. Ассемблер не присоединяет значения к имени psect.

Следующий пример показывает некоторые исполняемые инструкции, помещаемые в psect text и некоторые байты данных, помещаемые в psect data.

**psect text, global**

**alabel:**

**ld hl,astring**

**call putit**

**ld hl,anotherstring**

**psect data, global**

**astring:**

**defm 'A string of chars'**

**defb 0**

**anotherstring:**

**defm 'Another string'**

**defb 0**

**psect text**

**putit:**

**ld a,(hl)**

**or a**

**ret z**

**call outchar**

**inc hl**

**jr putit**

Обратите внимание, не смотря на то, что два блока кода в psect text разделены блоком кода psect data, два блока psect text будут непрерывны после загрузки компоновщиком. Инструкция ld hl, anotherstring будет попадать метку putit: во время выполнения. Фактическое расположение в памяти двух psect будет определено компоновщиком. Смотрите [Руководство компоновщика](#_Справочник_по_компоновщику) для получения информации о том, как определяются адреса psect.

Метка, определенная в psect, считается перемещаемой, т. к. ее фактический адрес в памяти не определяется во время ассемблирования. Обратите внимание, что это не так, если метка находится в (неназванной) psect по умолчанию, или в psect, объявленной абсолютной (см. [Описание псевдооперации PSECT](#_PSECT) ниже). Любые метки, объявленные в абсолютной psect, будут абсолютными, адрес которых определяется ассемблером.

Перемещаемые выражения могут свободно сочетаться в выражениях.

### Расширенные коды условий

Ассемблер распознает несколько дополнительных кодов условий, перечисленных в [Таблице 6-5](#t65) на странице 133.

Таблица 6‑5 Расширенные коды условий

|  |  |  |
| --- | --- | --- |
| **Код** | **Значение** | **Эквивалент** |
| **alt** | Арифметическое меньше чем | m |
| **llt** | Логическое меньше чем | c |
| **age** | Арифметическое больше или равно | p |
| **lge** | Логическое больше или равно | nc |
| **di,ei** | Используются после ld a,i для тестирования состояния флага разрешения прерывания - разрешены или запрещены соответственно. |  |

### Директивы ассемблера

Директивы ассемблера или псевдооперации, используются аналогично кодам операций, но или не генерируют код или генерируют неисполняемый код, т.е. байты данных. Директивы перечислены в [Таблице 6-6](#t66) на странице 134 и подробно описаны ниже.

Таблица 6‑6 Директивы ZAS (псевдооперации)

|  |  |
| --- | --- |
| **Директива** | **Назначение** |
| **GLOBAL** | Сделать символы доступными для других модулей или дать ссылку на символы других модулей |
| **END** | Завершение ассемблирования |
| **PSECT** | Объявление или возобновление секции программы |
| **ORG** | Установить счетчик адреса |
| **EQU** | Определить значение символа |
| **DEFL** | Определить или переопределить значение символа |
| **DEFB, DB** | Определить байтовую константу(ы) |
| **DEFW** | Определить слово(а) |
| **DEFF** | Определить действительную константу(ы) |
| **DEFM** | Определить сообщение |
| **DEFS, DS** | Зарезервировать место в памяти |
| **IF, COND** | Условное ассемблирование |
| **ELSE** | Альтернативное условное ассемблирование |
| **ENDC** | Завершение условного ассемблирования |
| **MACRO** | Определение макроса |
| **ENDM** | Завершение определения макроса |
| **LOCAL** | Определить локальные метки |
| **REPT** | Повторение блока с кодом n раз |
| **IRP** | Повторение блока с кодом из списка |
| **IRPC** | Повторение блока с кодом из списка символов |
| **SIGNAT** | Определение сигнатуры функции |

#### GLOBAL

Директива GLOBAL объявляет список символов, которые, если они определены в текущем модуле, становятся общедоступными, в противном случае это ссылки на символы во внешних модулях. Пример:

**GLOBAL lab1,lab2,lab3**

#### END

Директива END необязательная, но если представлена, должна быть в самом конце программы. Она завершит блок. Если в качестве параметра представлено выражение, то выражение будет использоваться для определения начального адреса программы. Будет ли он использоваться зависит от компоновщика. Пример:

**END start\_label**

#### PSECT

Директива PSECT объявляет или возобновляет секцию программы. Она использует в качестве параметров имя и необязательный список разделенных запятой флагов. Допустимые флаги перечислены в [Таблице 6-7](#t67) на странице 135 и описаны ниже. Как только psect объявлен, он может быть возобновлен позже, просто указывая его имя как аргумент другой директивы psect. Флаги повторять не нужно.

Таблица 6‑7 Флаги PSECT

|  |  |
| --- | --- |
| **Флаг** | **Назначение** |
| **ABS** | Секция psect является абсолютной |
| **CLASS** | Определяет имя класса psect |
| **GLOBAL** | Секция psect является глобальным (по умолчанию) |
| **LOCAL** | Локальная psect |
| **OVRLD** | Секция psect будет перекрывать те же psect в других модулях |
| **PURE** | Секция psect должна быть только для чтения |
| **RELOC** | Начать psect на указанной границе |
| **SELECTOR** | Номер страницы переключаемого банка памяти |
| **SIZE** | Максимальный размер psect |

* Флаг ABS определяет текущую psect как абсолютную, т. е. начинающуюся в позиции 0. Это не означает, что вклад этого модуля в psect начнется с нуля, поскольку другие модули могут внести свой вклад в ту же самую psect.
* Флаг CLASS указывает имя класса для этой psect. Названия классов используются, чтобы разрешить локальным psect ссылаться на имя класса во время компоновки, поскольку на них нельзя ссылаться по их собственному имени. Имена классов также полезны в тех случаях, когда используется функция диапазона адресов компоновщика.
* Секция psect, определенная как GLOBAL, будет объединена с другими глобальными psect с тем же именем из других модулей во время компоновки. GLOBAL - значение по умолчанию.
* Секция psect, определенная как LOCAL, не будет объединена с другим локальными psect во время компоновки, даже если будут другие с тем же именем. Локальная psect не может иметь такое же имя, как у любой глобальной psect, даже в другом модуле.
* Во время выполнения содержимое psect, определенной как OVRLD, будет накладываться из каждого модуля, а не присоединяться. Флаг OVRLD в сочетании с ABS определяет действительно абсолютную psect, т.е. psect, в которой любые определенные символы являются абсолютными.
* Флаг PURE сообщает компоновщику, что эта psect не будет изменена во время выполнения и может поэтому, например, быть помещена в ПЗУ. Этот флаг имеет ограниченную полезность, поскольку он зависит от компоновщика и целевой системы.
* Флаг RELOC позволяет определить требование для выравнивания psect на определенной границе, например. RELOC = 100h указывает, что эта psect должна начинаться с адреса, который кратен 100h.
* Флаг SELECTOR вычисляется компоновщиком из адреса загрузки psect, или автоматически по правилу по умолчанию, или по явному правилу передаваемому с помощью опции -g компоновщику.
* Флаг SIZE позволяет определить для psect максимальный размер, например, SIZE=100h. Флаг SIZE позволяет указать максимальный размер для psect, например. SIZE = 100h. Это условие будет проверено компоновщиком после объединения psect из всех модулей.

Ниже приводятся некоторые примеры использования директивы PSECT:

**PSECT fred**

**PSECT bill,size=100h,global**

**PSECT joh,abs,ovrld,class=CODE**

#### ORG

Директива ORG изменяет значение счетчика адреса в текущей psect. Это означает, что адреса с помощью ORG устанавливаются относительно начала psect, которое не определено до момента компоновки.

Параметр ORG должен быть или абсолютным значением или значением, ссылающимся на текущую psect. В любом случае текущий счетчик адреса будет установлен равным значению параметра. Например:

**ORG 100h**

Чтобы использовать директиву ORG для установки счетчика адреса в абсолютное значение, необходимо использовать наложенную psect:

**psect absdata, abs, ovrld**

**org addr**

где addr - абсолютный адрес.

#### EQU

Эта псевдооперация определяет символ и приравнивает его значение к выражению. Например,

**assembly EQU 123h**

где addr - абсолютный адрес.

Идентификатору ассемблера присваивается значение 123h. EQU является корректной только, если символ не был ранее определен. Операнд может также быть именем регистра, в этом случае символ станет синонимом регистра.

#### DEFL

DEFL (определение метки) идентична EQU за исключением того, что она может использоваться для переопределения символа.

#### DEFB, DB

DEFB и DB идентичны и используются, чтобы инициализировать память как байты. Параметр - список выражений, каждое из которых будет собрано в один байт. Также в качестве параметра DEFB и DB могут принять многосимвольную строку. Каждый символ строки будет помещен в одну ячейку памяти.

Если значение выражения будет слишком большим, чтобы вписаться в ячейку памяти возникнет ошибка, например, если значение 1020 задано как параметр DB.

Примеры:

**alabel: DB 'X',1,2,3,4,"A string",0**

#### DEFW

DEFW действует аналогично DEFB, за исключением того, что ассемблирует выражения в слова. Если значение выражения будет слишком большим, чтобы вписаться в слово возникнет ошибка.

Пример:

**DEFW -1, 3664H, 'A', 3777Q**

#### DEFF

DEFF инициализирует 4 байта памяти как вещественные числа. Каждое число будет сохранено в 32 разрядном формате HI-TECH C. Например:

**pi: DEFF 3.14159**

**DEFF 3.3,3e10,-23**

#### DEFS, DS

Эта директива резервирует, но не инициализирует, ячейки памяти. Единственный параметр - число резервируемых байтов. Примеры:

**alabel: DEFS 23 ; Резервирует 23 байта в памяти**

**xlabel: DEFS 2+3 ; Резервирует 5 байтов в памяти**

#### IF, COND, ELSE и ENDC

Эти директивы реализуют условное ассемблирование. IF и COND идентичны. Параметр IF должен быть абсолютным выражением. Если оно не будет равно нулю, то код после него до следующего соответствующего ELSE будет ассемблирован. Если выражение равно нулю, то код до следующего соответствующего ELSE будет пропущен.

ELSE инвертирует смысл условной компиляции, в то время как ENDC завершает блок условного ассемблирования. Пример:

**IF CPM**

**call 5**

**ELSE**

**call os\_func**

**ENDC**

В этом примере, если CPM отличен от нуля, будет ассемблирована первая инструкция вызова, а не вторая. С другой стороны, если CPM будет равен нулю, будет ассемблирован второй вызов, а первый не будет. Блоки условного ассемблирования могут быть вложены.

#### MACRO и ENDM

Эти директивы предусматривают определение макросов. Директиве MACRO должно предшествовать имя макроса и при необходимости последующий, разделенный запятыми список формальных параметров. Если используется макрос, имя макроса должно быть использовано таким же образом, как и машинный код операции, за которым следует список аргументов для замены формальных параметров.

Например:

**print MACRO string**

**psect data**

**999: db string,'$'**

**psect text**

**ld de,999b**

**ld c,9**

**call 5**

**ENDM**

При использовании этот макрос будет расширяться до 3 инструкций в теле макроса, при этом формальные параметры заменяются аргументами. Таким образом:

**print 'hello world'**

расширяется до:

**psect data**

**999: db 'hello world','$'**

**psect text**

**ld de,999b**

**ld c,9**

**call 5**

Символ "&" может использоваться для разграничения аргумента, используемого при кодировании макроса, что позволяет объединять параметры макроса с другим текстом, но удаляется в фактическом расширении макроса.

Символы "&" не должны использоваться, если запятые и пробелы разграничивают аргумент, но символ "&" следует использовать в начале и в конце, если другой разделитель не доступен.

Оператор NUL может использоваться в макросе для проверки макроса. Комментарий может быть подавлен при расширении макроса (таким образом, сохраняя место в хранилище макросов), открыв комментарий с помощью двойной точки с запятой (;;).

#### LOCAL

Директива LOCAL позволяет определять уникальные метки для каждого расширения данного макроса. Любые символы, перечисленные после директивы LOCAL, будут иметь уникальный ассемблерный символ, заменяющий их при расширении макроса. Например:

**copy MACRO source,dest,count**

**LOCAL nocopy**

**push af**

**push bc**

**ld bc,source**

**ld a,b**

**or c**

**jr z,nocopy**

**push de**

**push hl**

**ld de,dest**

**ld hl,source**

**ldir**

**pop hl**

**pop de**

**nocopy: pop bc**

**pop af**

**ENDM**

при расширении будет включать сгенерированную уникальную метку ассемблера вместо **nocopy**. Например, **copy (recptr), buf, (recsize)** расширится до:

**push af**

**push bc**

**ld bc,(recsize)**

**ld a,b**

**or c**

**jr z,??0001**

**push de**

**push hl**

**ld de,buf**

**ld hl,(recptr)**

**ldir**

**pop hl**

**pop de**

**??0001: pop bc**

**pop af**

При повторном вызове метка **nocopy** будет расширена в **??0002**.

#### REPT

Директива REPT временно определяет неименованный макрос, а затем расширяет его несколько раз, как определено его аргументом. Например:

**REPT 3**

**ld (hl),0**

**inc hl**

**ENDM**

будет расширен

**ld (hl),0**

**inc hl**

**ld (hl),0**

**inc hl**

**ld (hl),0**

**inc hl**

#### IRP и IRPC

Директивы IRP и IRPC действуют аналогично REPT. Однако вместо того, чтобы повторять блок постоянное число раз, он повторяется один раз для каждого элемента из списка аргументов. В случае IRP список является обычным списком аргументов макросов, в случае IRPC, это - каждый символ из одного параметра. При каждом повторении аргумент заменяет один формальный параметр.

Например:

**IRP string,<'hello world',13,10>,'arg2'**

**LOCAL str**

**psect data**

**str: db string,'$'**

**psect text**

**ld c,9**

**ld de,str**

**call 5**

**ENDM**

будет расширен

**psect data**

**??0001: db 'hello world',13,10,'$'**

**psect text**

**ld c,9**

**ld de,??0001**

**call 5**

**psect data**

**??0002: db 'arg2','$'**

**psect text**

**ld c,9**

**ld de,??0002**

**call 5**

Обратите внимание, что локальные метки и угловые скобки используются таким же образом, как и в обычных макросах.

Директива IRPC аналогична, за исключением того, что она заменяет один символ за раз из строки символов, отличных от пробела. Например:

**IRPC char,ABC**

**ld c,2**

**ld e,'char'**

**call 5**

**ENDM**

будет расширен

**ld c,2**

**ld e,'A'**

**call 5**

**ld c,2**

**ld e,'B'**

**call 5**

**ld c,2**

**ld e,'C'**

**call 5**

#### SIGNAT

Эта директива используется для сопоставления значение 16 разрядной сигнатуры с меткой. Во время связывания компоновщик проверяет, что все подписи, определенные для конкретной метки, одинаковы и выдает ошибку, если они не равны. Директива SIGNAT используется компилятором HI-TECH C для проверки во время компоновки прототипов функций и соглашений о вызовах.

Используйте директиву SIGNAT, если вы хотите писать ассемблерные подпрограммы, вызываемые из C. Например:

**SIGNAT \_fred,8192**

свяжет значение сигнатуры **8192** с символом **\_fred**. Если в любом объектном файле присутствует другое значение сигнатуры для **\_fred**, компоновщик сообщит об ошибке.

### Макро-вызовы

При вызове макроса список параметров должен быть разделен запятыми. Если в параметре требуется включить запятую (или другой разделитель, например пробел), то для цитирования параметра можно использовать угловые скобки (< и >). Кроме того, для цитирования одного символа можно использовать восклицательный знак (!). Знак, непосредственно следующий за восклицательным знаком, передается в аргумент макроса, даже если он обычно является индикатором комментария.

Если параметру будет предшествовать знак процента (%), то параметр будет вычислен как выражение и передан как десятичное число, а не как строка. Это полезно, если бы вычисление параметра в теле макроса привела бы к другому результату.

### Элементы управления ассемблером

Элементы управления ассемблером могут включаться в источник ассемблера для управления такими вещами, как формат листинга. Каждый элемент управления ассемблером начинается с символа звездочки (\*). Эти ключевые слова не имеют никакого значения в любом другом месте программы. За некоторыми ключевыми словами следует параметр.

Список ключевых слов приведен в [Таблице 6-8](#t68) на странице 141, и каждый из них описан ниже.

Таблица 6‑8 Управление ассемблером ZAS

|  |  |
| --- | --- |
| **Имя управления** | **Назначение** |
| **\*EJECT** | Начать новую страницу в листинге |
| **\*HEADING string** | Определить подзаголовок в листинге |
| **\*INCLUDE file** | Включить текст из другого файла |
| **\*LIST on|off** | Активировать или деактивировать листинг |
| **\*TITLE string** | Определить заголовок в листинге |

#### \*EJECT

Элемент управления \*EJECT заставляет начать новую страницу быть в листинге. Обнаружение в источнике символа Ctrl-L (прогон страницы) также вызовет новую страницу.

#### \*HEADING string

Элемент управления ассемблером \*HEADING определяет подзаголовок листинга. Подзаголовок отображается в верхней части каждой страницы, но может время от времени изменяться. Вот пример:

**\*heading Initialisation Phase.**

#### \*INCLUDE file

Этот элемент управления ассемблера может использоваться для включения текста из другого исходного файла. Например:

**\*include frame.inc**

#### \*LIST on|off

Эта директива включает и выключает листинг. Когда листинг отключен, сгенерированный код или исходный код, который его производит, не появляется в файле листинга.

Примеры:

**\*List on**

**\*List off**

#### \*TITLE string

Это ключевое слово управления определяет заголовок, который появляется сверху каждой страницы листинга. В файле заголовок может быть установлен только один раз. Например:

**\*title Heater Control Program**

# Справочник по компоновщику и утилитам

## Введение

HI-TECH C включает в себя перемещающий ассемблер и компоновщик, чтобы разрешить отдельную компиляцию исходных файлов C. Это означает, что программа может быть разделена на несколько исходных файлов, каждый из которых может быть доведен до приемлемого размера для удобства редактирования и компиляции, затем каждый исходный файл может быть скомпилирован отдельно и, наконец, все объектные файлы, связаны друг с другом в один исполняемый файл программы.

В этой главе описывается теория и использование компоновщика. Однако, обратите внимание, что в большинстве случаев нет необходимости использовать компоновщик напрямую, поскольку драйверы компилятора (HPD или командной строки) автоматически вызовут компоновщик со всеми необходимыми аргументами. Использование компоновщика напрямую непросто и должно быть предпринято только теми, у кого есть хорошее знание компилятора и компоновки в целом.

Если абсолютно необходимо использовать компоновщик напрямую, лучший способ начать - скопировать аргументы компоновщика, созданные драйвером компилятора, и изменить их соответствующим образом. Это обеспечит запуск необходимого модуля и наличие аргументов.

Отметим также, что компоновщик, поставляемый с HI-TECH C, является общим для множества компиляторов для нескольких разных процессоров. Не все функции, описанные в этой главе, применимы ко всем компиляторам.

## Перемещение и секции psect

Основная задача компоновщика состоит в том, чтобы объединить несколько перемещаемых объектных файлов в один. Объектные файлы, как говорят, являются перемещаемыми, так как файлы имеют достаточную информацию, позволяющую скорректировать любые ссылки на программы или адреса данных (например, адрес функции) в файле в соответствии с тем, где файл в конечном счете будет находится в памяти после процесса компоновки. Таким образом, файл считается перемещаемым. Перемещение может принять две основные формы. Перемещение по имени, т.е. перемещение по конечному значению глобального символа, или перемещение psect, т.е. перемещение по начальному адресу определенной секции кода, например секции кода, содержащей фактические выполняемые инструкции.

## Секции программ

Любой объектный файл может содержать байты, которые будут сохранены в памяти в одной или нескольких секциях программы, которые будут упоминаться как psect. Эти секции psect представляют некоторые логически сгруппированные байты кода в программе. В общем случае, компилятор произведет код в трех основных типах psect, хотя в каждом из них будет несколько разных типов. Три основных вида - секции psect text, содержащие исполняемый код, секции psect data, содержащие инициализируемые данные и секции psect bss, содержащие неинициализированные, но зарезервированные данные.

Различие между секциями data и bss можно проиллюстрировать, рассмотрев две внешние переменные, одна инициализирована значением 1, и другая не инициализирована. Первая будет помещена в секцию psect data и вторая в psect bss. Секция psect bss всегда очищается до нулей при запуске программы, таким образом вторая переменная будет инициализирована во время выполнения к нолю. Однако, первая займет место в файле программы и обеспечит ее инициализированное значение 1 при запуске. Можно изменить значение переменной в psect data во время выполнения, однако лучше не делать этого, так как это приводит к более последовательному использованию переменных и допускает прерываемые и размещаемые в ПЗУ программы.

Дополнительные сведения о конкретных psect, используемых конкретным компилятором, содержатся в соответствующей главе, относящейся к конкретной машине.

## Локальные psect

Большинство секций psect являются глобальными, то есть они называются одинаковыми именами во всех модулях, и любая ссылка в любом модуле на глобальный psect будет ссылаться на тот же самый psect, как и любая другая ссылка. Некоторые секции psect являются локальными, что означает, что они локальны только для одного модуля и будут считаться отдельными от любой другой psect, даже с тем же именем, в другом модуле. На локальные psect можно ссылаться только во время компоновки по имени класса, которое является именем, связанным с одним или несколькими psect с помощью директивы CLASS= в коде ассемблера.

## Глобальные символы

Компоновщик обрабатывает только те символы, которые были объявлены как глобальные для ассемблера. На уровне источника C это означает все имена, имеющие внешний класс хранения и которые не объявлены как статичные. Эти символы могут быть упомянуты модулями кроме того, в котором они определены. Задача компоновщика состоит в том, чтобы сопоставить определение глобального символа со ссылками на него. Другие символы (локальные символы) передаются через компоновщик в файл символов, но не обрабатываются компоновщиком.

## Адреса ссылки и загрузки

Компилятор имеет дело с двумя типами адресов, адресами ссылки и загрузки. Как правило, адрес ссылки секции psect - адрес по которому она доступна во время выполнения. Адрес загрузки, который может не совпасть с адресом ссылки, является адресом, в котором секция psect начинается в выходном файле (шестнадцатеричный или двоичный файл и т.д.). В случае процессора 8086 адрес ссылки примерно соответствует смещению в сегменте, в то время как адрес загрузки соответствует физическому адресу сегмента. Адрес сегмента - это адрес загрузки, разделенный на 16.

Другие примеры в которых адреса ссылки и загрузки различны. Инициализированные данные psect data, которые копируются из ПЗУ в ОЗУ при запуске, что может быть изменено во время выполнения. Переключаемая psect text, сопоставленная с физическим (== load) адресом на виртуальный (== link) адрес во время выполнения.

Точный способ использования адресов ссылки и загрузки в значительной степени зависит от используемого компилятора и модели памяти.

## Функционирование

Команда компоновщика принимает следующую форму:

**hlink[[18]](#footnote-18) опции файлы ...**

**Опции** - это ноль или более опций компоновщика, каждая из которых каким-то образом изменяет поведение компоновщика. **Файлы** - один или несколько объектных файлов, и ноль или больше имен библиотек. Опции, распознаваемые компоновщиком, перечислены в [Таблице 7-1](#t71) на странице 145 и описаны в следующих параграфах.

Таблица 7‑1 Опции компоновщика

|  |  |
| --- | --- |
| **Опция** | **Эффект** |
| **-A*class=low-high,...*** | Указать диапазон адресов для класса |
| **-C*x*** | Опции графа вызовов |
| **-C*psect=class*** | Определите название класса для глобальной psect |
| **-C*baseaddr*** | Создать выходной двоичный файл, начинающийся по *baseaddr* |
| **-D*class=delta*** | Указать значение смещения класса |
| **-D*symfile*** | Создать файл символов старого стиля |
| **-E*errfile*** | Записывать сообщения об ошибках в *errfile* |
| **-F** | Создать файл .OBJ только с записями символов |
| **-G*spec*** | Указать расчет для селектора сегментов |
| **-H*symfile*** | Создать файл символов |
| **-H+*symfile*** | Создать расширенный файл символов |
| **-I** | Игнорировать неопределенные символы |
| **-J*num*** | Установить максимальное количество ошибок до отмены |
| **-K** | Предотвращение наложения функций auto и областей параметров |
| **-L** | Сохранять элементы перемещения в файле .OBJ |
| **-LM** | Сохранять элементы перемещения сегмента в файле .OBJ |
| **-N** | Сортировка таблицы символов в map файле по адресам |
| **-Nc** | Сортировка таблицы символов в map файле по адресам классов |
| **-Ns** | Сортировка таблицы символов в map файле по пространственным адресам |
| **-M*mapfile*** | Создать карту памяти ссылок в названном файле |
| **-O*outfile*** | Указать имя выходного файла |
| **-P*spec*** | Определить адреса и расположение psect |
| **-Q*processor*** | Определить тип процессора (только по формальным причинам) |
| **-S** | Список запрещенных символов в файле символов |
| **-S*class=limit[,bound]*** | Определение ограничения адресов и начала границы для классов psect |
| **-U*symbol*** | Предварительный ввод символов в таблицу как неопределенных |
| **-V*avmap*** | Использовать файл avmap для создания файла символов в формате Avocet |
| **-W*warnlev*** | Установить уровень предупреждений (от -10 до 10) |
| **-W*width*** | Установить ширину map файла (>10) |
| **-X** | Удаление любых локальных символов из файла символов |
| **-Z** | Удаление несущественных локальных символов из файла символов |

### Числа в опциях компоновщика

Несколько опций компоновщика требуют указания адресов памяти или размеров. Синтаксис для всех аналогичен. По умолчанию число интерпретируется как десятичное значение. Чтобы принудительно интерпретировать число как шестнадцатеричное, в конце должен быть добавлен символ 'H', например, 765FH будет рассматриваться как шестнадцатеричное число.

### -A*class=low-high,...*

Обычно секции psect связываются в соответствии с информацией, предоставляемой опцией -P (см. ниже), но иногда желательно иметь класс psect, связанный с более чем одним несмежным диапазоном адресов. Эта опция позволяет указать ряд диапазонов адресов для класса. Например:

**-ACODE=1020h-7FFEh,8000h-BFFEh**

определяет, что класс CODE должен быть связан в заданные диапазоны адресов. Обратите внимание, что вклад в psect из одного модуля не может быть разделен, но компоновщик попытается упаковать каждый блок из каждого модуля в диапазоны адресов, начиная с первого указанного.

При наличии нескольких идентичных непрерывных диапазонов адресов они могут быть указаны с помощью счетчика повторений, например

**-ACODE=0-FFFFhx16**

определяет, что есть 16 смежных диапазонов каждый размером 64 Кбайт, начинающиеся с ноля. Хотя диапазоны являются непрерывными, код не выходят за границы 64 Кбайт. Счетчик повторений задается как символ "x" или "\*" после диапазона, за которым следует счетчик.

### -C*x*

Эти опции позволяют управлять информацией о графе вызовов, которая может быть включена в файл распределения памяти, произведенный компоновщиком. Опция -CN удаляет информацию о графе вызовов из файла распределения памяти. Опция -CC включают только критические пути графа вызовов. Вызов функции, который отмечен с "\*" в полном графе вызовов, находится на критическом пути, и только эти вызовы включены, когда используется опция -CC. Граф вызовов создается только для процессоров и моделей памяти, которые используют скомпилированный стек.

### -C*psect=class*

Эта опция позволяет связать psect с определенным классом. Обычно, она не требуется в командной строке, так как классы указаны в объектных файлах.

### -D*class=delta*

Эта опция позволяет указать значение **delta** для секций psect, которые являются членами указанного класса. Значение delta должно быть числом и представляет собой количество байтов на адресную единицу объектов внутри секций psect. Большинству секций psect этот параметр не требуется, поскольку они определяются значением delta.

### -D*symfile*

Используйте этот параметр для создания файла символов старого стиля. Символьный файл в старом стиле представляет собой файл ASCII, где каждая строка имеет адрес ссылки символа, за которым следует имя символа.

### -E*errfile*

Сообщения об ошибках компоновщика записываются в стандартный поток ошибок (дескриптор файла 2). В DOS нет удобного способа для перенаправления в файл (драйверы компилятора будут перенаправить стандартный поток ошибок, если перенаправляется стандартный вывод). Эта опция заставит компоновщик записать все сообщения об ошибках в указанный файл вместо экрана, который является стандартным местом назначения ошибок.

### -F

Обычно, компоновщик произведет объектный файл, который содержит байты с кодом программы и данных и информацию о символах. Иногда необходимо создать объектный файл только для символов, который может быть использован снова при последующем запуске компоновщика для предоставления значений символов. Опция -F подавит байты данных и кода из выходного файла, оставляя только записи символов.

Эта опция может использоваться при создании нескольких шестнадцатеричных файлов для ситуаций, когда программы содержатся в различных устройствах памяти расположенных по разным адресам. Файлы для одного устройства компилируются с использованием этой опции компоновщика для создания объектного файла только для символов. Затем он связывается с файлами для другого устройства. Затем процесс может быть повторен для других файлов и устройств.

### -G*spec*

При связывании программ с использованием сегментированных или коммутируемых с помощью банков секций psect существует два способа, которыми компоновщик может назначать адреса сегментам или селектора для каждого сегмента. Сегмент определяется как смежная группа секций psect, в которой каждая секция psect в последовательности имеет как адрес ссылки, так и адрес загрузки, объединенная с предыдущей секцией psect в группе. Адрес сегмента или селектор сегмента - это значение, производное от типа перемещения сегмента, которое обрабатывается компоновщиком.

По умолчанию селектор сегмента формируется путем деления начального адреса загрузки сегмента на квант перемещения сегмента, который основан на значении RELOC=, заданном для секций psect на уровне ассемблера. Это подходит для кода реального режима 8086, но не для защищенного режима или некоторых механизмов с переключением банков. В этом случае опция -G используется для указания метода вычисления селектора сегментов. Аргументом для -G является строка, подобная:

**A/10h-4h**

где A представляет адрес загрузки сегмента, и / представляет деление. Это означает, "Взять адрес загрузки psect, разделить на 10 hex, затем вычесть 4". Эта форма может быть изменена, заменив A на N, / на \* (для обозначения умножения), и - на +, чтобы добавить, а не вычесть константу. Символ N заменяется порядковым номером сегмента, который распределяется компоновщиком. Например:

**N\*8+4**

означает "взять номер сегмента, умножить на 8, а затем добавить 4". Результатом является селектор сегмента. В этом конкретном примере будут выделены селектора сегментов в последовательности 4, 12, 20, ... для определенных номеров сегментов. Это было бы целесообразно при компиляции для защищенного режима 80286, где эти селекторы будут представлять записи локальной таблицы дескрипторов LDT (**L**ocal **D**escriptor **T**able).

### -H*symfile*

Этот параметр даст указание компоновщику генерировать файл символов. Необязательный аргумент symfile указывает файл для присвоения файлу символов. Имя файла по умолчанию - l.sym.

### -H+*symfile*

Эта опция дает указание компоновщику генерировать расширенный файл символов, который помимо стандартного файла символов предоставляет имена классов, связанные с каждым символом, и секции сегментов, в которой перечислены каждое имя класса и область памяти, которую он занимает. Этот формат рекомендуется, если код должен быть запущен вместе с отладчиком. Необязательный аргумент symfile указывает файл для получения файла символов. Имя файла по умолчанию - l.sym.

### -J*errcount*

Компоновщик прекратит обрабатывать объектные файлы после определенного количества ошибок (кроме предупреждений). Это число по умолчанию равняется 10, но опция -J позволяет изменить это значение.

### -K

Для компиляторов, которые используют скомпилированный стек, компоновщик будет пытаться и накладывать функции auto и области параметров, пытаясь уменьшить общий объем ОЗУ. Для целей отладки эту функцию можно отключить с помощью этой опции.

### -I

Обычно невозможность разрешения ссылки на неопределенный символ является неустранимой (fatal) ошибкой. Использование этого параметра приведет к тому, что неопределенные символы будут обрабатываться как предупреждения (warning).

### -L

Когда компоновщик производит выходной файл, он обычно не сохраняет информацию о перемещении, так как файл теперь абсолютный. В некоторых случаях дальнейшее "перемещение" программы будет выполнено во время загрузки, например, при запуске .EXE файла в DOS или .PRG файла в TOS (операционная система компании Atari). Для этого требуется, чтобы некоторые сведения о том, какие адреса требуют перемещения, сохранялись в объектном (а затем в исполняемом файле). Опция -L генерирует в выходной файл одну нулевую запись перемещения для каждой записи перемещения при вводе.

### -LM

Аналогична опции, описанной выше, при этом сохраняются записи перемещения в выходной файл, но только перемещения сегментов. В частности, она используется для создания .EXE файлов для запуска в DOS.

### -M*mapfile*

Эта опция заставляет компоновщик генерировать распределение памяти связей в названном файле, или на стандартном выходе, если имя файла опущено. Формат файла распределения памяти показан в [Разделе 7.9](#_Карты_распределения_памяти) на странице 152.

### –N, -Ns и  -Nc

По умолчанию таблица символов в карте распределения ссылок будет отсортирована по имени. Параметр -N приведет к численной сортировке на основе значений символов. Параметры -Ns и -Nc работают аналогично, за исключением того, что символы группируются по типу пространства (Space) или типу класса (Class).

### -O*outfile*

Этот параметр позволяет задавать имя выходного файла компоновщика. Имя выходного файла по умолчанию l.obj. Использование этой опции переопределяет значение по умолчанию.

### -P*spec*

Секции psect соединяются и получают адреса на основе информации, предоставленной компоновщику с помощью опции -P. Аргумент опции -P состоит в основном из разделенных запятыми последовательностей следующим образом:

**-Ppsect=lnkaddr+min/ldaddr+min,psect=lnkaddr/ldaddr, ...**

Существует несколько вариантов, но по существу каждая секция psect перечисляется с нужными адресами ссылки и загрузки и минимальным значением. Все значения могут быть опущены, в этом случае будут применяться значения по умолчанию, в зависимости от предыдущих значений.

Минимальному значению (**min**) предшествует знак +, если присутствует. Он устанавливает минимальное значение для адреса ссылки или загрузки. Адрес будет вычислен, как описано ниже, но если он меньше минимального, то он будет установлен равным минимуму.

Адреса ссылки и загрузки - или числа, как описано выше, или имена других секций psect, или классов или специальных символов. Если адрес ссылки будет опущен, адрес ссылки psect будет получен из вершины предыдущей секции psect, например

**-Ptext=100h,data,bss**

В этом примере psect **text** связан по адресу 100 hex (по умолчанию его адрес загрузки по умолчанию тот же). Секция psect **data** будет связана (и загружена) по адресу, который является 100 hex плюс длина psect **text**, округленный при необходимости, если psect **data** имеет значение RELOC= value, связанное с ней. Аналогично, psect **bss** будет соединена с psect **data**.

Если адрес загрузки полностью опущен, он по умолчанию совпадает с адресом ссылки. Если указан символ наклонной черты вправо (/), но после него адрес отсутствует, адрес загрузки будет сцеплен с предыдущей psect, например

**-Ptext=0,data=0/,bss**

заставит секции **text** и **data** иметь адрес ссылки равный нулю, секция **text** будет иметь адрес загрузки 0, и секция **data** будет адрес загрузки, начинающийся после конца секции **text**. Секция psect **bss** будет сцеплена с **data** как по адресу ссылки, так и адресу загрузки.

Адрес загрузки может быть заменен символом точки (.). Она говорит компоновщику установит адрес загрузки этой секции psect равным адресу ее ссылки. Адрес ссылки или загрузки могут также быть именем другой (уже связанной) psect. Это явно сцепит текущую psect с ранее указанной psect, например.

**-Ptext=0,data=8000h/,bss/. -Pnvram=bss,heap**

Этот пример показывает **text** равным нулю, **data**, связанна в 8000h, но загружена после **text**, **bss** связана и загружена в 8000h плюс размер **data**, и **nvram** и **heap** сцеплены с **bss**. Отметьте здесь использование двух опций -P. Несколько опций -P обрабатываются по порядку.

Если для определения диапазонов адресов класса использовались опции -A, то имя этого класса может использоваться вместо адресов ссылки и загрузки, и место будет найдено в одном из диапазонов адресов. Например:

**-ACODE=8000h-BFFEh,E000h-FFFEh**

**-Pdata=C000h/CODE**

Это свяжет данные в C000h, но найдет, что пространство загружает его в диапазоны адресов, связанные с CODE. Если достаточно большое пространство не будет доступно, возникнет ошибка. Обратите внимание , что в этом случае psect **data** все еще будут собраны в один непрерывный блок, тогда как другие секции psect в классе CODE будут распределены в диапазоны адресов везде, где они поместятся. Это означает, что, если в классе CODE есть две или более psect, они могут быть смешаны в диапазонах адресов.

Любые секции psect, выделенные опцией -P, будут иметь диапазон адресов загрузки, вычитаемый из любых диапазонов адресов, заданных опцией -A. Это позволяет указать диапазон с помощью опции -A, не зная заранее, какой объем нижней части диапазона, например, потребуется для других psect.

### -Q*processor*

Эта опция позволяет определить тип процессора. Это исключительно для информации, помещаемой в файл карты распределения памяти. Параметром этой опции является строка, описывающая процессор.

### -S

Эта опция препятствует включению в файл символов информации о символах, произведенных компоновщиком. Информация о сегменте по прежнему включена.

### -S*class=limit[,bound]*

Класс секций psect может иметь верхний предел адреса, связанный с ним. Следующий пример устанавливает границу максимального адреса секций psect класса CODE на единицу меньшей чем 400h.

**-SCODE=400h**

Обратите внимание, что для установки верхнего предела psect он должен быть задан в коде ассемблера (с установленным флагом LIMIT=on в директиве psect).

Если используется аргумент bound (граничный), класс секций psect начинается с адреса кратного bound. Этот пример помещает класс FAPCODE секций psect по адресу кратному 1000h, но с верхним пределом адреса 6000h:

**-SFARCODE=6000h,1000h**

### -U*symbol*

Этот параметр вводит указанный символ в таблицу символов компоновщика как неопределенный символ. Это полезно для полного связывания с библиотеками или для связывания модуля с библиотекой, где упорядочение организовано так, что по умолчанию будет связан более поздний модуль.

### -V*avmap*

Чтобы создать файл символов формата Avocet, компоновщику необходимо предоставить файл карты, чтобы он мог сопоставить имена psect с идентификаторами памяти Avocet. Файл avmap обычно поставляется с компилятором или автоматически создается драйвером компилятора при необходимости.

### -W*num*

Опция -W может использоваться, чтобы установить уровень предупреждений, в диапазоне от -9 до 9, или ширина файла карты распределения памяти, при значениях >= 10.

Опция -W9 подавит все предупреждающие сообщения. -W0 значение по умолчанию. При установке уровня предупреждений равным -9 (-W-9) будут выдаваться самые полные предупреждающие сообщения.

### -X

С помощью этой опции локальные символы могут быть подавлены в файле символов. Глобальные символы всегда появляются в файле символов.

### -Z

Некоторые локальные символы генерируются компилятором и не представляют интерес для отладки. Эта опция будет подавлять из файла символа все локальные символы, которые имеют форму одного алфавитного символа, за которым следует цифра. Набор букв с которых может начинаться несущественные символы, в настоящее время "klfLSu". Опция -Z очистит любые локальные символы, начинающиеся с одной из этих букв, а затем цифры.

## Вызов компоновщика

Компоновщик называется HLINK, и обычно находится в подкаталоге BIN каталога установки компилятора. Он может быть вызван без параметров, и в этом случае он запросит ввод из стандартного ввода. Если стандартным вводом будет файл, никакие подсказки не будут распечатаны. Этот способ вызова обычно полезен, если количество параметров HLINK велико. Даже если список файлов слишком длинный, чтобы поместиться в одной строке, могут быть включены строки продолжения, поместив наклонную черту влево ('\') в конце предыдущей строки. Этим способом могут быть заданы команды HLINK почти неограниченной длины. Например, командный файл компоновщика названный X.LNK и содержащий следующий текст:

**-Z -OX.OBJ -MX.MAP \**

**-Ptext=0,data=0/,bss,nvram=bss/. \**

**X.OBJ Y.OBJ Z.OBJ C:\HT-Z80\LIB\Z80-SC.LIB**

Может быть передан компоновщику одной из следующих команд:

**hlink @x.lnk**

**hlink <x.lnk**

## Карты распределения памяти

Файл карты содержит информацию, относящуюся к перемещению psect и адресов, назначенных символам в пределах этих psect. Разделы в файле карты: Первый - это копия командной строки, используемая для вызова компоновщика. За ней следует номер версии объектного кода в первом связанном файле и тип машины. При необходимости можно получить информацию о графике вызовов, зависящую от выбранной модели процессора и памяти. Затем перечисляются все объектные файлы, которые были связаны, вместе со своей информацией о psect. Библиотеки перечислены с каждым модулем в библиотеке. Раздел TOTALS суммирует секции psect из объектных файлов. Раздел SEGMENTS обобщает основные группы памяти. Обычно это отображает использование ОЗУ и ПЗУ. Имена сегментов являются производными от имени первого psect в сегменте.

Наконец (не показанная в примере) таблица символов, в которой каждый глобальный символ отображается со связанной с ним psect и адресом ссылки.

Linker command line:

-z -Mmap -pvectors=00h,text,strings,const,im2vecs -pbaseram=00h \

-pramstart=08000h,data/im2vecs,bss/.,stack=09000h -pnvram=bss,heap \

-oC:\TEMP\l.obj C:\HT-Z80\LIB\rtz80-s.obj hello.obj \

C:\HT-Z80\LIB\z80-sc.lib

Object code version is 2.4

Machine type is Z80

Name Link Load Length Selector

C:\HT-Z80\LIB\rtz80-s.obj

vectors 0 0 71

bss 8000 8000 24

const FB FB 1 0

text 72 72 82

hello.obj text F4

C:\HT-Z80\LIB\z80-sc.lib

powerup.obj vectors 71 71 1

TOTAL Name Link Load Length

CLASS CODE

vectors 0 0 72

const FB FB 1

text 72 72 89

CLASS DATA

bss 8000 8000 24

SEGMENTS Name Load Length Top Selector

vectors 000000 0000FC 0000FC 0

bss 008000 000024 008024 8000

### Информация о графе вызовов

Граф вызовов создается для типов микросхем и моделей памяти, использующих скомпилированный стек, а не аппаратный стек для облегчения передачи параметров между функциями и автоматическими переменными, определенными в функции. При использовании скомпилированного стека, функции не реентерабельные, так как функция использует фиксированную область памяти для своих локальных объектов (параметры/автоматические переменные). Функция с именем foo, например, будет использовать символы как ?\_foo для параметров и ?a\_foo для автоматических переменных. Компиляторы, такие как PIC, 6805 и V8 используют скомпилированные стеки. Компилятор 8051 использует скомпилированный стек в малых и средних моделях памяти. В графе вызовов отображается информация о размещении компоновщиком параметров функций и автоматических переменных. Типичный граф вызовов может выглядеть примерно так:

Call graph:

\*\_main size 0,0 offset 0

\_init size 2,3 offset 0

\_ports size 2,2 offset 5

\* \_sprintf size 5,10 offset 0

\* \_putch

INDIRECT 4194

INDIRECT 4194

\_function\_2 size 2,2 offset 0

\_function size 2,2 offset 5

\*\_isr->\_incr size 2,0 offset 15

В графе показаны вызываемые функции и использование памяти (ОЗУ) функцией для собственных локальных объектов. В приведенном выше примере символ **\_main** соотносится с функцией **main**. Он отображается в левой части графа вызовов. Это означает, что это корень дерева вызовов. Код времени выполнения имеет директиву ассемблера **FNROOT**, которая определяет это. Поле size после имени указывает на число параметров и автоматических переменных, соответственно. Здесь, **main()** не принимает параметров и не определяет автоматические переменные. Поле offset - это смещение, при котором параметры и автоматические переменные функции были размещены с начала области памяти, используемой для этой цели. Код времени выполнения содержит директиву **FNCONF**, которая указывает компилятору, где должны находиться параметры psect и автоматические переменные. Эта память будет показана в файле карты под именем **COMMON**.

Функция **main()** вызывает функцию, названную init. Эта функция использует в общей сложности два байта параметров (это могут быть два char или один int, это не важно), и имеет три байта переменных auto. Эти числа - общее количество байтов памяти, используемых функцией. Если функции передается двухбайтовый int, но это выполняется с помощью регистра, то два байта не включаются в это общее количество. Поскольку **main()** не использовала память локальных объектов, смещение памяти init() по-прежнему равняется 0.

Функция **init** сама вызывает другую функцию, названную **ports**. Эта функция использует два байта параметров и еще два байта переменных auto. Так как **ports()** вызывает **init()**, ее локальные переменные не могут перекрываться с локальными значениями для **init()**, поэтому смещение равно 5, что означает, что локальные объекты **ports()** были помещены сразу же после **init()**.

Функция **main** также, вызывает **sprintf()**. Поскольку функция **sprintf** не активна одновременно с **init()** или **ports()**, их локальные объекты могут быть перекрыты, и offset следовательно установлен равным 0. Функция **sprintf()** вызывает функцию putch, но эта функция не использует память для параметров (в качестве аргумента передается char, который по-видимому, передается через регистр) или локальных переменных, таким образом, size и offset равны нулю и не напечатаны.

Функция **main()** также косвенно вызывает другую функцию, используя указатель функции. В графе это обозначено двумя записями **INDIRECT**. Следующее число - значения сигнатуры функций, которые теоретически могут быть вызваны косвенным вызовом. Это число вычислено из типа параметров и возврата функций, которые косвенно может вызвать указатель. Имена любых функций, имеющие эти значения сигнатуры, перечислены под записями **INDIRECT**. Их включение не означает, что они будут вызваны (не существует способа определить это), но теоретически их можно вызвать.

Последняя строка показывает другую функцию, имя которой расположено в крайне левом месте графа вызовов. Это подразумевает, что это - корень другого дерева графа вызовов. Это - функция прерывания, которая не вызывается из кода, но которая автоматически вызывается, когда происходит разрешенное прерывание. Эта процедура прерывания вызывает функцию **incr()**, которая условно показана в графе символом "**->**", сопровождаемым именем вызванной функции вместо того, чтобы показать эту функцию с отступом на следующей строке. Это делается всякий раз, когда вызывающая функция не принимает параметров, и не определяет каких-либо переменных.

Строки на графе, отмеченные звездочкой (\*), являются функциями, находящимися на критическом пути с точки зрения использования ОЗУ. Например, в приведенном выше примере (**main()** является простым примером) рассмотрим функцию **sprintf**. Она использует большое количество локальной памяти, и если вы можете каким-то образом переписать ее так, чтобы она использовал меньше локальной памяти, это уменьшило бы использование ОЗУ всей программы. Функции **init** и **ports** имеют локальную память, перекрываемую с **sprintf()**, поэтому уменьшение размера локальной памяти этих функций не влияет на использование ОЗУ программы. Используемая ими память может быть увеличена до тех пор, пока общий размер памяти, используемой этими двумя функциями, не будет превышать объем памяти **sprintf()**, не требуя дополнительной памяти, используемой программой. Поэтому, если вам нужно уменьшить объем оперативной памяти, используемой программой, смотрите на помеченные функции.

Если при просмотре графа вызовов вы заметили, что параметры функции и области auto перекрываются (т. е. ?a\_foo был помещен по тому же адресу, что и ?\_foo, например), проверьте, что вы действительно вызвали функцию в программе. Если компоновщик действительно не видел вызванную функцию, то она перекрывает эти области памяти, поскольку они не требуется. Это является следствием способности компоновщика перекрывать локальные области памяти функций, которые не являются активными в одно и то же время. После вызова функции как параметрам, так и объектам auto будут назначены уникальные адреса.

При написании процедуры, вызывающей код C из ассемблера, необходимо включить соответствующие директивы ассемблера для обеспечения того, чтобы компоновщик увидел вызов функции C.

## Библиотеки

Программа библиотекаря, LIBR, имеет функцию объединения нескольких объектных файлов в один файл, известный как библиотека. Существует несколько причин для объединения таких объектных модулей.

* Меньше файлов для связывания
* Более быстрый доступ
* Используется меньше места на диске

Чтобы сделать концепцию библиотеки полезной, необходимо, чтобы компоновщик обрабатывал модули в библиотеке иначе, чем объектные файлы. Если объектный файл указан компоновщику, он будет связан с последним связанным модулем. Между тем, модуль из библиотеки будет связан только в том случае, если он определяет один или несколько ранее известных, но не определенных символов компоновщику. Таким образом, модули из библиотеки будут связаны только при необходимости. Так как выбор связываемых модулей выполняется на первом проходе компоновщика, и поиск в библиотека осуществляется линейным способом, можно использовать порядок модулей в библиотеке для создания специальных эффектов при связывании. Об этом будет сказано позже.

### Формат библиотеки

Модули в библиотеке в основном только объединяются, но в начале библиотеки сохраняется каталог модулей и символов в библиотеке. Так как этот каталог меньше, чем сумма модулей, редактор связей может ускорить поиск в библиотеке, так как ему нужно читать только каталог, а не все модули на первом проходе. На втором проходе ему нужно читать только те модули, которые необходимы, осуществляя поиск поверх других. Это все минимизирует дисковый ввод-вывод при компоновке.

Нужно отметить, что формат библиотеки приспособлен исключительно к объектным модулям и не является механизмом архивации общего назначения, используемым некоторыми другими дистрибутивами компиляторов. Это имеет преимущество в том, что формат может быть оптимизирован в направлении ускорения процесса связывания.

### Использование библиотек

Программа библиотекаря называется LIBR, и формат команды выглядит следующим образом:

**LIBR опции k file.lib file.obj ...**

Интерпретируя это, **LIBR** - имя программы, **опции** –ноль или более опций библиотекаря, **k** - ключевая буква, обозначающая, запрашиваемую функцию библиотекаря (замена, извлечение или удаление модулей, перечисление модулей или символов), **file.lib** является именем файла обрабатываемой библиотеки и **file.obj** - ноль или более имен объектных файлов.

Опции библиотекаря перечислены в [Таблице 7-2](#t72).

Ключевые буквы приведены в [Таблице 7-3](#t73).

Таблица 7‑2 Опции библиотекаря

|  |  |
| --- | --- |
| **Опция** | **Действие** |
| **-P*width*** | Определить ширину страницы |
| **-W** | Подавить не серьезные ошибки |

Таблица 7‑3 Ключевые буквы команд библиотекаря

|  |  |
| --- | --- |
| **Опция** | **Назначение** |
| **r** | замена модулей |
| **d** | удаление модулей |
| **x** | извлечение модулей |
| **m** | список имен модулей |
| **s** | список модулей с символами |

При замене или извлечении модулей, параметры file.obj - имена модулей, которые будут заменены или извлечены. Если параметры отсутствуют, то все модули в библиотеке будут заменены или извлечены соответственно. Добавление файла в библиотеку выполняется, запросив библиотекарь заменить его в библиотеке. Так как он отсутствует, модуль будет добавлен в библиотеку. Если используется ключ **r** и библиотека не существует, то она будет создана.

С ключевой буквой **d** названные объектные файлы будут удалены из библиотеки. В данном случае, отсутствие заданных имен объектных файлов является ошибкой.

Ключевые буквы **m** и **s** перечислят названные модули и, в случае использования ключевой буквы **s**, символы, определенные или упомянутые внутри (библиотекарем обрабатываются только глобальные символы). Как и с ключевыми буквами **r** и **x**, пустой список модулей означает все модули в библиотеке.

### Примеры

Вот некоторые примеры использования библиотекаря. Следующая команда перечисляет глобальные символы в модулях a.obj, b.obj и c.obj:

**LIBR s file.lib a.obj b.obj c.obj**

Эта команда удаляет объектные модули a.obj, b.obj и 2.obj из библиотеки file.lib:

**LIBR d file.lib a.obj b.obj 2.obj**

### Предоставление аргументов

Поскольку часто необходимо передать LIBR много объектных файлов в виде параметров, а командные строки CP/M и MS-DOS ограничены 127 символами, LIBR принимает команды из стандартного ввода, если не заданы параметры командной строки. Если стандартный ввод будет присоединен к консоли, то LIBR выведет приглашение. Может быть введено несколько строк, используя наклонную черту влево в качестве символа продолжения в конце строки. Если стандартный ввод будет перенаправлен из файла, то LIBR возьмет ввод от файла без приглашения. Например:

**LIBR**

libr> **r file.lib 1.obj 2.obj 3.obj \**

libr> **4.obj 5.obj 6.obj**

выполнит так же, как будто .obj файлы были введены в командной строке. Приглашение **libr>** было распечатано самой программой LIBR, остальной текст был набран в качестве входных данных.

**libr <lib.cmd**

Программа LIBR считает ввод из lib.cmd и выполнит команду, найденную там. Это позволяет передавать к LIBR команды практически неограниченной длины.

### Формат листинга

Запрос LIBR перечислить имена модуля просто выведет список имен по одному в каждой строке на стандартный вывод. Ключ **s** выведет к тому же, список символов после каждого имени модуля. Каждому символу будет предшествовать буква D или U, представляя определение или ссылку на символ соответственно. Опция -P может использоваться, чтобы определить ширину бумаги для этой операции. Например, **LIBR -P80 s file.lib** перечислит все модули в **file.lib** с их глобальными символами с выводом отформатированным для принтера или дисплея с 80 столбцами.

### Упорядочивание библиотек

Библиотекарь создает библиотеки с модулями в порядке, в котором они заданы в командной строке. При обновлении библиотеки порядок модулей сохраняется. Любые новые модули, добавляемые в библиотеку после ее создания, будут добавляться в конце.

Порядок следования модулей в библиотеке имеет важное значение для компоновщика. Если библиотека содержит модуль, который ссылается на символ, определенный в другом модуле в этой же библиотеке, модуль, определяющий символ, должен появиться после модуля, ссылающегося на символ.

### Сообщения об ошибках

Программа LIBR выводит различные сообщения об ошибках, большинство из которых являются серьезной ошибкой, а некоторые представляют собой безобидное явление, о котором, тем не менее, будет сообщаться, если не использовалась опция -W. В этом случае все предупреждающие сообщения будут подавлены.

## OBJTOHEX

Компоновщик HI-TECH способен к созданию простых двоичных файлов или объектных файлов в качестве выхода. Любой другой требуемый формат должен быть произведен, выполнив утилиту OBJTOHEX. Она позволяет преобразовать созданные компоновщиком объектные файлы во множество разных форматов, включая различные шестнадцатеричные форматы. Программа вызывается следующим образом:

**OBJTOHEX опции входной\_файл выходной\_файл**

Все опции необязательные. Если выходной файл опущен, то по умолчанию он будет l.hex или l.bin в зависимости от того, используется ли параметр -b. Входным файлом по умолчанию является l.obj.

Опции для OBJTOHEX перечислены в [Таблице 7-4](#t74) на странице 158. Где требуется адрес, его формат совпадает с HLINK.

### Спецификация контрольной суммы

Спецификация контрольной суммы позволяет автоматизированное вычисление контрольной суммы. Спецификация контрольной суммы принимает форму нескольких строк, каждая строка, описывает одну контрольную сумму. Синтаксис строки контрольной суммы следующий:

**addr1-addr2 where1-where2 +offset**

Все addr1, addr2, where1, where2 и offset являются шестнадцатеричными числами без обычного суффикса H. Такая спецификация говорит, что байты с addr1 до addr2 включительно должны быть суммированы, и сумма помещена в расположения начиная с where1 до where2 включительно. Для 8-разрядной контрольной суммы эти два адреса должны совпадать. Поскольку контрольная сумма сохраняет сначала младший байт, where1 должен быть меньше, чем where1 и наоборот. Элемент +offset является необязательным, но, если присутствует, значение offset будет использоваться для инициализации контрольной суммы. Иначе она будет обнулена. Например:

**0005-1FFF 3-4 +1FFF**

суммирует байты с 5 по 1FFFH включительно, затем добавить 1FFFH к сумме. 16 разрядная контрольная сумма будет помещена в расположения 3 и 4, младший байт в 3. Контрольная сумма инициализируется значением 1FFFH, чтобы обеспечить защиту от всех нулевых ПЗУ или ПЗУ неуместных в памяти. Во время выполнения проверки этой контрольной суммы будет добавлен последний адрес ПЗУ для проверки суммированием в контрольной сумме. Для рассматриваемого ПЗУ он должен быть 1FFFH. Однако, инициализируемое значение может использоваться в любой желаемой форме.

Таблица 7‑4 Опции OBJTOHEX

| **Опция** | **Значение** |
| --- | --- |
| **-A** | Произвести ATDOS выходной файл .ATX |
| **-B*base*** | Произвести вывод двоичного образа. Имя файла по умолчанию - l.obj |
| **-C*ckfile*** | Читать спецификацию контрольной суммы из *ckfile* или стандартного ввода |
| **-D** | Произвести файл .COD |
| **-E** | Произвести MS-DOS файл .EXE |
| **-F*fill*** | Заполнить неиспользованную память байтами со значением *fill*, по умолчанию - 0FFh |
| **-I** | Произвести файл Intel HEX с линейно адресуемыми расширенными записями. |
| **-L** | Передача информации о перемещениях в выходной файл (используется с .EXE файлами) |
| **-M** | Произвести файл Motorola HEX (в формате S19, S28 или S37) |
| **-N** | Произвести выходной файл для Minix |
| **-P*stk*** | Произвести выходной файл для Atari ST с дополнительным размером стека |
| **-R** | Включить информацию о перемещении в выходной файл |
| **-S*file*** | Записать файл символов в файл |
| **-T** | Произвести файл Tektronix HEX. -TE производит расширенный файл TekHEX |
| **-U** | Произвести файл в формате COFF |
| **-UB** | Произвести файл в формате UBROF |
| **-V** | Обратный порядок слов и длинных слов в выходном файле |
| **-x** | Создать файл в формате x.out |

## CREF

Утилита списка перекрестных ссылок CREF используется для форматирования необработанной информации о перекрестных ссылках, произведенной компилятором или ассемблером в отсортированный список. Файл с необработанными перекрестными ссылками создается компилятором с помощью параметра -CR. Ассемблер генерирует файл с необработанными перекрестными ссылками с помощью параметра -C (большинство ассемблеров) или при помощи директивы OPT CRE (модификации ассемблеров 6800). Общая форма команды CREF следующая:

**CREF опции файлы**

где **опции** - ноль или более опций, описанных ниже и **файлы** один или несколько необработанных файлов перекрестных ссылок. CREF использует опции, приведенные в [Таблице 7-5](#t75) на странице 159. Каждая опция более подробно описана в следующих разделах.

### -F*prefix*

Часто бывает необходимо исключить из перекрестной ссылки, перечисляющей любые символы, определенные в системном заголовочном файле, например, <stdio.h>. Опция -F позволяет определить префикс имени пути, который будет использоваться для исключения любых символов, определенные в файле, имя пути которого начинается с этого префикса. Например, опция '-F\' исключит любые символы из всех файлов с именем пути, начинающегося с '\'.

Таблица 7‑5 Опции CREF

| **Опция** | **Значение** |
| --- | --- |
| **-F*prefix*** | Исключить символы из файлов с именем пути или именем файла, начинающегося с ***prefix*** |
| **-H*heading*** | Определить заголовок (***heading***) файла листинга |
| **-L*len*** | Определить длину (***len***) страницы файла листинга |
| **-O*outfile*** | Определить имя файла (***outfile***) листинга |
| **-P*width*** | Установить ширину (***width***) листинга |
| **-S*stoplist*** | Прочитать файл ***stoplist*** и игнорировать любые перечисленные символы. |
| **-X*prefix*** | Исключить любые символы, начинающиеся с данного ***prefix*** |

### -H*heading*

Опция **-H** принимает строку в качестве аргумента, которая будет использоваться в качестве заголовка (*heading*) в листинге. Заголовок по умолчанию - название первой необработанной информации указанного файла.

### -L*len*

Определяет длину (*len*) страницы, на которой должен быть создан листинг, например, если листинг должен быть напечатан на странице содержащей 55 сток необходимо использовать опцию -L55. По умолчанию - 66 строк.

### -O*outfile*

Позволяет указать имя выходного файла (*outfile*). По умолчанию перечисление будет записано в стандартный вывод и может быть перенаправлено обычным способом. Альтернативно *outfile* может быть определен в качестве имени выходного файла.

### -P*width*

Эта опция позволяет указать ширину (*width*), к которой должно быть отформатировано перечисление, например, -P132 отформатирует перечисление для принтера с 132 столбцами. Значение по умолчанию составляет 80 столбцов.

### -S*stoplist*

Опция -S должен иметь в качестве своего аргумента имя файла, содержащего список символов, которые не будут перечислены в перекрестной ссылке. Могут использоваться несколько стоп-листов с помощью нескольких опций -S.

### -X*prefix*

Опция -X позволяет исключение символов из листинга, на основе префикса, заданного как аргумент -X. Например, если необходимо исключить все символы, начинающиеся с последовательности символов xyz, то необходимо использовать опцию -Xxyz. Если в последовательности символов появится цифра, то это будет соответствовать любой цифре в символе, например, -XX0 исключит любые символы, начинающиеся с буквы X и последующей цифры.

CREF примет подстановочные имена файлов и перенаправление ввода-вывода. Длинные командные строки могут быть предоставлены при вызове CREF без параметров и вводе командной строки в ответ на запрос cref>. Обратная косая черта в конце строки будет интерпретироваться как означающая, что далее будут следовать командные строки.

## MEMMAP

Программа MEMMAP индивидуальная для каждого процессора. Программа MEMMAP, которая появляется в каталоге \bin, будет соответствовать следующим критериям. XXmap.exe, где XX обозначает тип процессора. Здесь и далее мы будем иметь в виду XXmap.exe, используя MEMMAP, чтобы охватить все процессоры.

В конце компиляции и компоновки, HPD и компилятор командной строки производят сводку использования памяти. Однако, если компиляция выполняется на отдельных этапах, и компоновщик вызывается явно, эта информация о памяти не отображается. Программа MEMMAP считывает информацию, сохраненную в файле карты, и производит или сводку распределения адресов psect или карту памяти разделов программы, подобные показанным HPD и компилятором командной строки.

### Использование MEMMAP

Команда программы использования памяти имеет форму:

**MEMMAP опции файл**

Опции - ноль или больше опций MEMMAP, которые перечислены в [Таблице 7-6](#t76) на странице 160. Файл подразумевает имя фала карты. Только один файл карты может быть обработан MEMMAP.

Таблица 7‑6 Опции MEMMAP

|  |  |
| --- | --- |
| **Опция** | **Действие** |
| **-P** | Печатать карты используемых psect |
| **-W*wid*** | Определяет ширину печати адресов |

#### -P

Поведение по умолчанию MEMMAP состоит в том, чтобы произвести карту памяти сегмента. Этот вывод подобен распечатанному HPD и компилятором командной строки после компиляции и компоновки. Это поведение может быть изменено при помощи опции -P. Она вынуждает распечатывать карту использования psect. Вывод в этом случае будет подобен показанному пунктом Memory Usage Map (Карта использования памяти) в меню Utility (Утилиты) HPD или при использовании опции PSECTMAP компилятором командной строки.

#### -Wwid

Ширина, на которую печатаются адреса, может быть изменена с помощью опции -W. Ширина по умолчанию определяется по диапазону адресов процессора. В зависимости от типа используемого процессора определяется ширина по умолчанию распечатанного адреса, например, процессор с адресующий до 64 Кбайт будет иметь ширину по умолчанию 4. В то время как процессор адресующий более 64 Кбайт может иметь значение по умолчанию 6 цифр.

# Отладчик уровня исходного кода Lucifer

*Lucifer* представляет собой удаленный отладчик уровня исходного кода для использования с компиляторами HI-TECH C. Он состоит из программы, которая работает на хост-машине (обычно MS-DOS или UNIX) и связывается с микроконтроллерами построенными на процессорах Z80, Z180, 64180 или NSC800 через последовательную линию.

Хост-программа предоставляет пользовательский интерфейс, включая отображение исходного кода, дизассемблирование, отображение памяти и т.д. Целевая система должна иметь логику для чтения и записи памяти и регистров, а также реализации одного шага. С каждой версией Lucifer предоставляется небольшая программа, которую можно скомпилировать и поместить в ПЗУ целевой системы для реализации этих функций. Стандартная программа хоста настроена для связи с этой программой ПЗУ через последовательную линию.

## Использование Lucifer

Чтобы использовать Lucifer, вам нужно, чтобы компилятор сгенерировал файл символов с именами символов, номерами строк и имя файла включенных символов. Это можно сделать, используя опцию ZC -G. Если вы используете опцию -H, вы получите файл символов, который может быть использован Lucifer, но который не содержит информацию уровня исходного кода.

Также вы можете использовать HPDZ для создания файлов HEX и символов, подходящих для использования с Lucifer. Каталог LUCIFER содержит файл демонстрационного проекта *luctest.prj*, который можно использовать в качестве руководства по компиляции ваших собственных программ с помощью HPDZ. Смотрите [Главу 3](#_Использование_HPDZ_3) "*Использование HPDZ*" для получения дополнительной информации. Основной необходимой опцией является **Options/Source level debug info** (Опции/информацией об отладке на уровне исходного кода). Lucifer может быть вызван из пункта меню **Run/Download...** (Выполнение/Загрузка...). В противном случае, как только вы создали файлы HEX и символов, вызовите Lucifer следующим образом:

**lucz80 -sSPEED -pCOMn test**

Если вы используете систему MS-DOS, *COMn* должен быть COM1 или COM2. Lucifer получит доступ к стандартному последовательному порту, адресуемому как любой порт. Для UNIX нужно просто указать имя последовательного порта, подключенного к вашему целевому устройству, например, "-p/dev/tty006". Скорость передачи по умолчанию 38400 и для MS-DOS и для UNIX. Последовательный порт по умолчанию - *COM1* для MS-DOS и */dev/ttya* для UNIX. Опции -s (скорость) и -p (порт) могут использоваться для доступа к портам отличным от назначенных по умолчанию. Для UNIX /dev можно не указывать в имени устройства, и оно будет автоматически добавлено, таким образом "-ptty0", и "-p/dev/tty0" получит доступ к одному и тому же устройству. Например, под MS-DOS, **lucz80 -s9600 -pCOM2** получит доступ к COM2 при скорости передачи 9600.

Новые значения по умолчанию скорости и порта могут быть установлены, используя переменную окружения *LUCZ80\_ARGS*. LUCZ80\_ARGS может определить любую смесь допустимых в Lucifer '-' опций, кроме опций с именами файлов. Например, если вы хотите, чтобы опции по умолчанию определяли 4800 бод порта COM2, добавьте следующую строку в свой файл AUTOEXEC.BAT:

**SET LUCZ80\_ARGS=-s4800 -pCOM2**

В дополнение к опциям скорости и порта Lucifer принимает два необязательных параметра в порядке появления, имя файла символов для использования и имя .HEX или .BIN файла для загрузки. Если файл загрузки не определен, Lucifer будет автоматически искать .SYM, .HEX и .BIN файлы с тем же базовым именем как у заданного ими символов. Таким образом команда: **lucz80 test** будет автоматически обнаруживать и использовать test.sym и test.hex или test.bin. Если вы не хотите автоматически загружать файл HEX или BIN, задайте вашему файлу символов другое базовое имя для вашего HEX файла.

При загрузке двоичных файлов Lucifer обычно запрашивает адрес загрузки. При загрузке непосредственно из командной строки можно изменить этот запрос, добавив опцию -*Baddr[:end]* в командной строке LUCZ80. Например, если вы хотите загрузить файл *test.bin* по адресу $2000, можно использовать команду:

**lucz80 -b2000 test**

Дополнительное значение *:end* - адрес, в котором должна быть завершена загрузка. Например, если вы хотите загрузить первые байт $2000 test.bin начиная с адреса $4000 до адреса $6000 используйте команду:

**lucz80 -b4000:6000 test**

Lucifer должен объявить о себе, а затем попытаться связаться с целью. При успешном завершении он распечатывает сообщение, отправленное целью, идентифицируя себя, например:

**Z80/64180 Lucifer v3.20, RST = CF**

Затем Lucifer выводит на экран подсказку : и ожидает команду. Для получения списка команд введите ? и нажмите Return. Обратите внимание, что *все команды должны быть в нижнем регистре*.

## Имена символов в выражениях

Там, где команды Lucifer принимают числовые значения или адреса, в качестве параметров могут использоваться имена символов, имена регистров и номера строк. Символы должны быть введены в точно в том же виде, как они были определены в исходном коде. Обратите внимание, что Lucifer не может получить доступ к автоматическим переменным или параметрам по имени. Там, где требуется выражение, оно может иметь формы приведенные в [Таблице 8-1](#t81) на странице 162.

Таблица 8‑1 Формы выражения Lucifer

|  |  |
| --- | --- |
| **Форма выражения** | **Пример** |
| **symbol\_name** | main |
| **symbol+hexnum** | barray+20 |
| **$hexnum** | $2000 |
| **:linenumber** | :10 |
| **regname** | A5 |

По умолчанию в команде *b* (точка останова) любое десятичное число интерпретируется как номер строки. Однако в команде *u* (дизассемблировать), по умолчанию любое число будет интерпретироваться, как шестнадцатеричное число, определяющее адрес. Эти предположения всегда могут переопределяться при помощи префиксов доллара или двоеточия. При вводе символа не нужно вводить предшествующее подчеркивание, ожидаемое компилятором C. Однако при распечатывании символов отладчик будет всегда распечатывать подчеркивание. В месте, где ожидается символ может использоваться имя любого регистра.

### Автоматические переменные и параметры

В Lucifer к автоматическим переменным и параметрам нельзя получить доступ по имени. Чтобы исследовать содержание автоматической переменной или параметра, лучшим подходом является дизассемблирование (с помощью команды *u*) строку кода, ссылающуюся на переменную, и вывести соответствующую ячейку памяти (например, *sp+4*).

## Список команд Lucifer

Lucifer распознает команды, перечисленные в [Таблице 8-2](#t82) на странице 163.

Таблица 8‑2 Список команд Lucifer

|  |  |
| --- | --- |
| **Команда** | **Значение** |
| **B** [строка/адрес] | Установка или отображение точек останова |
| **C** | Отображение инструкции по адресу PC |
| **D** [адрес[адрес]] | Отображение содержимого памяти |
| **E** функция/файл | Исследование исходного кода C |
| **G** [адрес] | Начало выполнения |
| **I** | Переключение режима трассировки инструкций |
| **L** файл | Загрузка шестнадцатеричного файла |
| **M** адрес [знач1] [знач2…] | Изменение памяти |
| **P** | Переключение приглашения ввода |
| **Q** | Выход в операционную систему |
| **R** [контрольная точка] | Удаление точки останова |
| **S** | Шаг на одну строку |
| **T** | Трассировка одной инструкции |
| **U** [адрес] | Дизассемблирование машинных команд |
| **W** файл адрес длина | Загрузка двоичного файла |
| **X** [рег1 знач1 [рег2 знач2 ...]] | Исследование или изменение регистров |
| **@** тип [ссылка] выражение | Отображение переменных C |
| **.** [контрольная точка] | Установка точки останова и переход |
| **;** [строка] | Отображение от исходной строки |
| **=** | Отображение следующей страницы источника |
| - | Отображение предыдущей страницы источника |
| **/** [строка] | Поиск строки в исходном файле |
| **!** Команда | Выполнение команды DOS |

### Команда B: Установка или отображение точек останова

Команда **b** используется для установки и отображения точек останова. Если после команды b отсутствует выражение, будет выведен список всех установленных в настоящее время точек останова. Если выражение предоставлено, точка останова будет установлена в строке или по определенному адресу. При попытке установить существующую точку останова, или ввода выражения, которое не может понять Lucifer, на экран будет выведено соответствующее сообщение об ошибке. Примечание: по умолчанию любое определенное десятичное число интерпретируется как номер строки. Если вы хотите определить абсолютный адрес, снабдите его префиксом знак доллара. Например:

: **b 10**

Set breakpoint at \_main+$28

:

Точки останова могут также включать список команд Lucifer разделенных символами точка с запятой, которые будут выполняться, после встречи с точкой останова. Это позволяет создать контрольные точки, в которых происходит остановка выполнения, вывод на экран значения и затем возобновляется выполнение. Например, команда **:** **b** **10** @f **pi;g** создает точку останова, которая останавливает выполнения, выводит на экран значение глобальной переменной *pi* и затем продолжает выполнение.

### Команда C: Отображение инструкции в PC

Команда **C** используется для вывода на экран ассемблерной инструкции и исходной строки C, адресуемой текущим значением счетчик команд (PC). Это полезно, если вы используете другие команды Lucifer и не совсем уверены, на что в программе указывает счетчик команд. Например:

: **c**

10: printf("answer = %d\n",j);

\_main+$22 MOV r0,\_j

### Команда D: Отображение содержимого памяти

Команда **D** используется, чтобы вывести на экран шестнадцатеричный дамп содержания памяти в целевой системе. Если выражение не определено, выводятся 16 байтов от адреса, достигнутого последней командой **D**. Если определен один адрес, выводятся 16 байтов от заданного адреса. Если определены два адреса, отображается содержание памяти от первого адреса до второго адреса. В качестве адреса дампа могут использоваться символы, номера строк, имена регистров или абсолютные адреса памяти.

### Команда E: Исследование исходного кода C

Команда **E** используется для исследования исходного кода функции или файла C. Если задано имя функции, Lucifer определит местоположение исходного файла, содержащего требуемую функцию и выведет на экран с места расположенного чуть выше начала функции. Если задано имя файла, Lucifer выведет на экран требуемый файл начиная с 1 строки. Например:

: **e main**

2:

3:int value, result; 4:

5:main()

6:{

7: scanf("%d",&value);

8: result = (value << 1) + 6;

9: printf("result = %d\n",result);

10:}

:

### Команда G: Начало выполнения

Команда **G** используется, чтобы начать выполнение кода на целевой системе. Если выражение отсутствует, то после команды **G** выполнение начнется с текущего значения PC (счетчика команд). Если выражение присутствует, выполнение начнется с заданного адреса. Выполнение будет продолжаться, до достижения контрольной точки, или прерывания пользователем с помощью Ctrl-C. После достижения контрольной точки выполнение может продолжаться от этого места, используя команды **G**, **S** и **T**.

### Команда I: Переключение режима трассировки инструкций

Команда **I** используется для переключения в режим трассировки команд. Если режим трассировки команд включен, каждая инструкция отображается, прежде чем она будет выполнена при трассировке шага всей строки C командой **S**. Например, с отключенной трассировкой инструкций, шаг ведет себя следующим образом:

:**s**

result = 20 Stepped to

10:}

:

С включенной трассировкой инструкций шаг будет вместо этого вести себя следующим образом:

:**s**

\_memtest+30H push r4

\_memtest+32H mov r0,#04A2H

\_memtest+36H push r0

\_memtest+38H fcall \_printf

\_memtest+3CH adds r7,#4

result = 20

Stepped to

10:}

:

Обратите внимание, что библиотечная функция printf() не трассировалась и, таким образом, правильно выполнялась на полной скорости.

### Команда L: Загрузка шестнадцатеричного файла

Команда **L** используется для загрузки файлов в целевую систему. Lucifer правильно обрабатывает S-записи объектных файлов формата Motorola, файлы Intel HEX и бинарные образы.

### Команда M: Изменение памяти

Команда **M** используется для записи одного или более значений или строк ASCII в память по указанному адресу. Эта команда принимает форму: ***M адрес знач1 [знач2...]***, где адрес это адрес для записи, а все следующие аргументы являются значениями или строками для записи в память. Строки могут использовать или одинарные или двойные кавычки. Например: : **m buf "hello" 13 10 'world' 0**

### Команда P: Переключение приглашения ввода

Команда **P** используется для переключения приглашения ввода. Если приглашение ввода включено, Lucifer выведет на экран подсказку "Target wants input:" (Цель ожидает ввод) когда целевая программа выполняет функции ввода (*gets()*, *scanf()*, и др.). Если приглашение ввода отключено, приглашение ввода остается в целевой программе.

### Команда Q: Выход в операционную систему

Команда **Q** используется для выхода из Lucifer в операционную систему. Примечание: команда **Q** не останавливает целевую систему (т.е. монитор Lucifer, работающий на целевой системе), таким образом, можно повторно войти в Lucifer, не инициализируя повторно цель.

### Команда R: Удаление контрольной точки

Команда **R** используется для удаления точек останова, которые были установлены с помощью команды **B**. Если параметры отсутствуют, пользователю предлагают каждую точку останова поочередно. Например:

: **r**

Remove \_main+$28 ? **y**

Remove \_main+$44 ? **n**

Remove \_test ? **n**

: **r main+$44**

Removed breakpoint \_main+$44

:

### Команда S: Шаг на одну строку

Команда **S** используется для выполнения трассировки одной строки C или ассемблерного кода. Например:

: **s**

Stepped to

7: scanf("%d", &value);

: **s**

Target wants input: 7

Stepped to

8: result = (value << 1) + 6;

: **s**

Stepped to

9: printf("result = %d\n",result); :s

result = 20

Stepped to

10:}

:

Это обычно реализуется путем выполнения нескольких шагов трассировки машинных инструкций, и поэтому может быть довольно медленным. Если Lucifer может определить, что в строке нет вызовов функций или управляющих структур (*break*, *continue* и т. д.), Он установит временную точку останова на следующей строке и выполнит строку на полной скорости. При пошаговой трассировке машинных команд команда шага будет выполнять вызовы внешних подпрограмм и библиотечных функций на полной скорости. Это позволяет избежать медленного процесса пошагового выполнения сложных библиотечных процедур, таких как printf(). Нормальные библиотеки ввода-вывода при пошаговой трассировке, используя команду **S**, работают правильно. Если информация о номере строки отсутствует, например, внутри библиотеки, команда **S** становится шагом ассемблера, как команда **T**.

### Команда T: Трассировка одной инструкции

Команда **T** используется для трассировки одной машинной инструкции на целевой системе. Текущее значение PC (счетчик команд) используется в качестве адреса выполняемой инструкции. После выполнения инструкции отображается следующая инструкция и содержание всех регистров.

### Команда U: Дизассемблирование машинных команд

Команда **U** дизассемблирует объектный код в памяти целевой системы. Например:

: **u**

9: printf("result = %d\n", result);

\_memtest+30H push r4,r5

\_memtest+32H mov r0,#04A2H

\_memtest+36H push r0

\_memtest+38H fcall \_printf

\_memtest+3CH adds r7,#6

Если присутствует выражение, дизассемблирование начинается с предоставленного адреса. Если адрес отсутствует, дизассемблирование начинается с инструкции, где закончилось последнее дизассемблирование. Если доступна таблица символов для дизассемблируемой программы, дизассемблер автоматически преобразует адреса в объектном коде в символы. Если доступен исходный код для дизассемблируемой программы C, также отображаются строки C, соответствующие каждой группе инструкций. Примечание: по умолчанию все указанные значения будут интерпретироваться как абсолютные адреса. Если вы хотите определить номер строки, снабдите его префиксом двоеточие.

### Команда W: Загрузка двоичного файла

Команда **W** используется для загрузки и записи блока целевой памяти в виде двоичного файла. Эта команда использует три параметра: имя файла, начальный адрес и длину. Значения начального адреса и длины находятся в шестнадцатеричном формате. Например, если монитор Lucifer находится в ПЗУ по адресам от $7000 до $7FFF в целевой системе, он может быть загружен в двоичный файл с помощью команды:

: **w lucrom.bin 7000 1000**

........

Uploaded 4096 (0x1000) bytes to lucrom.bin

:

### Команда X: Исследование или изменение регистров

Команда **X** используется для исследования и изменения содержания регистров процессора целевой системы. Если параметры не заданы, регистры отображаются без изменения. Для изменения содержимого регистра, должны быть представлены два параметра: допустимое имя регистра и новое значение регистра. После присвоения нового значения регистру отображается содержание регистров.

### Команда @: Отображение переменных C

Команда @ используется для исследования содержания памяти, интерпретируемой как один из стандарта C типов. Форма команды следующий: @t/[\*]expr, где t - тип переменной, которая будет выведена на экран, \*, состоит из нуля или большего количества косвенных операторов (“\*” или “n\*”), и expr - адрес отображаемой переменной. [Таблица 8-3](#t83) на странице 168, показывает доступное варианты команды.

Например,

Отобразить переменную типа long longvar в шестнадцатеричном виде: **@lx longvar**

Отобразить символ, на который указывает указатель cptr: **@c \*cptr**

Разыменовать ihandle: указатель на указатель на беззнаковое целое: **@iu \*\*ihandle**

Таблица 8‑3 Варианты команд @ Lucifer

|  |  |  |
| --- | --- | --- |
| **Команда** | **Тип** | **Вывод** |
| @c | char | Символ и значение |
| @cu | unsigned char | десятичный |
| @cx | unsigned char | Шестнадцатеричный |
| @co | unsigned char | Восьмеричный |
| @i | int unsigned | Десятичный |
| @iu | int | Десятичный |
| @ix | unsigned int | Шестнадцатеричный |
| @io | unsigned int | Восьмеричный |
| @l | long | Десятичный |
| @lu | unsigned long | Десятичный |
| @lx | unsigned long | Шестнадцатеричный |
| @lo | unsigned long | Восьмеричный |
| @f | float | Десятичное |
| @np | near pointer | Символ+смещение |
| @p | pointer | Символ+смещение |
| @s | string | Строка символов |

После отображения переменной текущий адрес увеличивается на размер ее типа. Это, позволяет передвигаться по массивам, многократно нажимая Return. Онлайн-справка для команды @ может быть получена, введя ?@ в приглашении ":".

### Команда .: Установка контрольной точки и переход

Команда . используется, чтобы установить временную точку останова и продолжить выполнение с текущего значения PC (счетчика команд). Выполнение продолжается, до достижения любая точки останова или пользовательские прерывания пользователем с помощью Ctrl-C, после чего временная точка останова удаляется. Примечание: временная точка останова удаляется, даже если остановка выполнения происходит в другой точке останова или после прерывания. Если адрес точки останова не определен, команда . отобразит список активных точек останова.

: . 10

Target wants input: 7

result = 20

Breakpoint

10:}

main+$28 RET

:

### Команда ;: Отображение от исходной строки

Команда ; используется для отображения 10 строк исходного кода от указанной позиции в исходном файле. Если номер строки опущен, последняя страница выведенного на экран исходного кода отображена повторно. Например:

: **; 4**

4:

5: main() 6: {

7: scanf("%d",&value);

8: result = (value << 1) + 6;

9: printf("result = %d\n",result);

10:}

### Команда =: Отображение следующей страницы источника

Команда **=** используется для отображения следующих 10 строк исходного кода из текущего файла. Например, если последняя исходная строка, выведенная на экран была строкой 7, = выведет на экран 10 строк, начиная со строки 8.

### Команда -: Отображение предыдущей страницы источника

Команда **-** используется для отображения предыдущих 10 строк исходного кода из текущего файла. Например, если последняя отображенная страница начиналась в строке 15, - выведет на экран 10 строк, начиная со строки 5.

### Команда /: Поиск строки в исходном файле

Команда / используется для поиска в текущем исходном файле вхождения последовательности символов. Любой текст, введенный после / используется для поиска в исходном файле. Отображается первая строка источника, содержащие указанную строку. Если после символа / текст не введен, осуществляется поиск предыдущей строки. Поиск каждой строки начинается с точки, где закончился предыдущий, позволяя пользователю пошагово выполнять поиск всех вхождений строки в исходный файл.

: **/printf**

10: printf("Enter a number:");

: **/**

14: printf("Result = %d\n",answer);

: **/**

Can’t find printf

:

### Команда !: Выполнение команды DOS

Команда **!** используется для выполнения команды интерпретатором команд операционной системы, не выходя из Lucifer. Любой текст, введенный после **!** передается интерпретатору команд без изменений.

### Другие команды

В дополнение к упомянутым выше командам Lucifer интерпретирует любое допустимое десятичное число, введенное как номер исходной строки и попытается отобразить исходный код C для этой строки.

Нажатие Return, без ввода команды приведет к повторному выполнению предыдущей команды. В большинстве случаев команда возобновляется с места, где завершилась предыдущая. Например, если предыдущая команда была **d 2000**, нажатие Return будет иметь тот же эффект как команда **d 2010**.

## Ввод вывод пользователя в Lucifer

Стандартные версии подпрограмм консольного ввода-вывода *putch()*, *getch()*, *getche()* и *init\_uart()* настроены для автоматической работы с Lucifer. Код, загружаемый в Lucifer, может использовать стандартные подпрограммы ввода-вывода, такие как *printf()*, без каких-либо изменений в библиотеке. После завершения отладки кода вам нужно будет вставить в библиотеку консоли подпрограммы ввода-вывода, подходящие для вашего оборудования. Вы можете использовать файл getch.c в каталоге *SOURCES* в качестве отправной точки.

## Установка Lucifer на целевой компьютер

Для использования Lucifer в вашей целевой системе, необходимо скомпилировать монитор Lucifer и поместить его в ПЗУ. Если ваша система Z80 уже имеет монитор в ПЗУ, можно загрузить целевой код Lucifer в ОЗУ. В большинстве случаев вы сможете использовать поставляемую программу монитора Lucifer без значительных изменений.

### Изменение целевого кода

Как правило, изменения требуют только драйвер последовательного порта и код одного шага.

Предоставлены три версии целевого кода; *jtarget.c*, который сконфигурирован для платы JED STD 801, *ztarget.c*, который сконфигурирован для использования с универсальным Z80 с последовательным портом Z80-SIO и *targ180.c*, который установлен для использования с Z180 или 64180 процессорами, используя один из встроенных последовательных портов.

В этих исходных файлах имеются обширные комментарии, документирующие любые изменения, которые могут потребоваться для различных аппаратных средств.

# Сообщения об ошибках

В этой главе перечислены все возможные сообщения об ошибках компилятора HI-TECH C с объяснением каждой из них. Название приложений, которые могли вызвать ошибку, указаны в скобках, напротив сообщения об ошибке. В учебном разделе описываются функции каждого приложения.

'.' expected after '..' *(Синтаксический анализатор)*

('.' ожидается после '..')

Единственный контекст, в котором могут появиться две последовательные точки, является частью символа многоточия, который должен иметь 3 точки.

'case' not in switch *(Синтаксический анализатор)*

('case' не в switch)

Был встречен оператор case, но отсутствует содержащий его оператор switch. Оператор case может появляться только внутри тела оператора switch.

'default' not in switch *(Синтаксический анализатор)*

('default' не в switch)

Обнаружена метка с именем "default", но она не в операторе switch. Метка "default" допустима только в теле оператора switch.

( expected *(Синтаксический анализатор)*

(ожидается ' (' )

Здесь ожидалась открывающая скобка. Она должна быть первым токеном после зарезервированного слова while, for, if, do или asm.

) expected *(Синтаксический анализатор)*

(ожидается ' )' )

Здесь ожидалась закрывающая круглая скобка. Это может означать, что в выражении пропущена круглая скобка или есть другая синтаксическая ошибка.

\*: no match *(Препроцессор, Синтаксический анализатор)*

(\*: не совпадает)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

, expected *(Синтаксический анализатор)*

(ожидается ',' )

Здесь ожидалась запятая. Возможно она означает, что в списке объявления отсутствует запятая между двумя идентификаторами. Она также может означать, что непосредственно предшествующее имя типа имеет орфографическую ошибку и поэтому интерпретируется как идентификатор.

-s, too few values specified in \* *(Препроцессор)*

(-s, слишком мало значений, определенных в \*)

В опции -S неполный список значений к препроцессору. Этого не должно происходить, если препроцессор вызывается драйвером компилятора или HPD.

-s, too many values, \* unused *(Препроцессор)*

(-s, слишком много значений, \* не используется)

Было задано слишком много значений в опции препроцессора -S.

... illegal in non-prototype arg list *(Синтаксический анализатор)*

(… не допустимы в списке непрототипных аргументов )

Символ многоточия может отображаться только в качестве последнего элемента в списке аргументов с прототипом. Он не может отображаться отдельно и не может появляться после имен аргументов, не имеющих типов.

: expected *(Синтаксический анализатор)*

(ожидается ':' )

Двоеточие отсутствует в метке case или после ключевого слова "default". Это часто происходит, когда вместо двоеточия случайно вводится точка с запятой.

; expected *(Синтаксический анализатор)*

(ожидается ';' )

Здесь отсутствует точка с запятой. Точка с запятой используется в качестве терминатора во многих типах операторов, например, do .. while, return и т.д.

= expected *(Кодогенератор, Ассемблер)*

(ожидается '=' )

Здесь ожидался знак равенства.

#define syntax error *(Препроцессор)*

(Ошибка в синтаксисе #define)

Определение макроса имеет синтаксическую ошибку. Она может быть связана с именем макроса или формального параметра, которое не начинается с буквы или отсутствием закрывающей круглой скобки (')').

#elif may not follow #else *(Препроцессор)*

(#elif не может следовать за #else)

Если #else использовался после #if нельзя использовать #elif в том же условном блоке.

#elif must be in an #if *(Препроцессор)*

(#elif должно быть в #if)

#elif должно предшествовать соответствующая строка #if. Если соответствующая строка #if явно существует, проверьте наличие таких вещей, как дополнительные комментарии #endif или неправильно завершенные комментарии.

#else may not follow #else *(Препроцессор)*

(#else не может следовать за #else)

Может быть только один #else, соответствующий каждому #if.

#else must be in an #if *(Препроцессор)*

(#else должно быть в #if)

#else может использоваться только после соответствующего #if.

#endif must be in an #if *(Препроцессор)*

(#endif должно быть в #if)

Должен быть соответствующий #if для каждого #endif. Проверьте корректно ли количество #if.

#error: \* *(Препроцессор)*

Это ошибка, вызванная программистом. Существует директива, вызывающая преднамеренную ошибку. Обычно она используется для проверки во время компиляции и т. д.)

#if ... sizeof() syntax error *(Препроцессор)*

Препроцессор обнаружил синтаксическую ошибку в параметре sizeof в выражении #if. Вероятными причинами являются несогласованные круглые скобки и подобные вещи.

#if ... sizeof: bug, unknown type code \* *(Препроцессор)*

(#if ... sizeof: ошибка, неизвестный код типа \*)

Препроцессор допустил внутреннюю ошибку при вычислении выражения sizeof(). Проверьте, нет ли описателя неправильного типа.

#if ... sizeof: illegal type combination *(Препроцессор)*

(#if ... sizeof: недопустимая комбинация типа)

Препроцессор обнаружил недопустимое сочетание типа в аргументе для sizeof() в выражении #if. Недопустимые комбинации включают такие вещи, как "short long int".

#if bug, operand = \* *(Препроцессор)*

Препроцессор попытался вычислить выражение с помощью оператора, который не понимает. Это внутренняя ошибка.

#if sizeof() error, no type specified *(Препроцессор)*

В препроцессоре в выражении #if использовался sizeof(), но тип не был определен. Аргумент sizeof() в выражении препроцессора должен быть допустимым простым типом или указателем на простой тип.

#if sizeof, unknown type \* *(Препроцессор)*

Неизвестный тип был использован в препроцессоре sizeof(). Препроцессор может вычислять только sizeof() с базовыми типами или указателями на базовые типы.

#if value stack overflow *(Препроцессор)*

Препроцессор заполнял свой стек вычисления выражения в выражении #if. Упростите выражение - оно, вероятно, содержит слишком много скобок вложенных выражений.

#if, #ifdef, or #ifndef without an argument *(Препроцессор)*

( отсутствует аргумент в #if, #ifdef или #ifndef)

Директивы препроцессора #if, #ifdef и #ifndef должны иметь аргумент. Аргумент для #if должен быть выражением, в то время как аргумент #ifdef или #ifndef должен быть единственным именем.

#include syntax error *(Препроцессор)*

( синтаксическая ошибка в #include)

Недопустимый синтаксис аргумента filename для #include. Аргумент #include должен быть допустимым именем файла, заключенным в двойные кавычки ("") или угловые скобки (<>). Например:

#include "afile.h"

#include <otherfile.h>

Пробелы не должны включаться, а закрывающие кавычки или скобки должны присутствовать. В строке не должно быть ничего другого.

#included file \* was converted to lower case *(Препроцессор)*

(Включенный файл \* был преобразован в нижний регистр)

Имя файла #include должно быть преобразовано в нижний регистр, прежде чем его можно будет открыть.

] expected *(Синтаксический анализатор)*

(ожидается ']' )

В объявлении массива или в выражении с использованием индекса массива ожидалась закрывающая квадратная скобка.

{ expected *(Синтаксический анализатор)*

(ожидается '{' )

Здесь ожидалась открывающая фигурная скобка.

} expected *(Синтаксический анализатор)*

(ожидается '}' )

Здесь ожидалась закрывающая фигурная скобка.

a parameter may not be a function *(Синтаксический анализатор)*

(параметр не может быть функцией)

Параметр функции не может быть функцией. Это может быть указатель на функцию, так что, возможно, в объявлении опущен символ "\*".

absolute expression required *(Ассемблер)*

(требуется абсолютное выражение)

В этом контексте требуется абсолютное выражение.

add\_reloc - bad size *(Ассемблер)*

(add\_reloc - неправильный размер)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

ambiguous format name '\*' *(Cromwell)*

( неопределенный формат имени '\*')

Выходной формат, определенный Cromwell, неоднозначен.

argument \* conflicts with prototype *(Синтаксический анализатор)*

(аргумент \* конфликтует с прототипом)

Указанный аргумент (аргумент 1 является самым левым аргументом) этого объявления функции не согласуется с предыдущим прототипом этой функции.

argument -w\* ignored *(Компоновщик)*

(аргумент –w\* проигнорирован)

Аргумент опции -w компоновщика выходит за пределы допустимого диапазона. Для уровней предупреждения диапазон составляет от -9 до 9. Для ширины файла карты распределения памяти диапазон больше или равен 10.

argument list conflicts with prototype *(Синтактический анализатор)*

(список аргументов конфликтует с прототипом)

Список аргументов в определении функции не совпадает с предыдущим прототипом для этой функции. Убедитесь, что количество и типы аргументов одинаковы.

argument redeclared: \* *(Синтактический анализатор)*

(аргумент повторно объявлен: \*)

Указанный аргумент объявлен несколько раз в том же списке аргументов.

argument too long *(Препроцессор, Синтактический анализатор)*

(слишком длинный аргумент)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

arithmetic overflow in constant expression *(Генератор кода)*

(арифметическое переполнение в константном выражении)

При вычислении константного выражение генератором кода, получено значение, которое является слишком большим для типа выражения, например, попытка сохранить значение 256 в "char".

array dimension on \* ignored *(Препроцессор)*

(размерность массива в \* проигнорирована)

Размерность массива в параметре функции игнорируется, поскольку этот аргумент фактически преобразуется в указатель при его передаче. Таким образом, могут быть переданы массивы любого размера.

array dimension redeclared *(Синтактический анализатор)*

(размерность массива объявлена повторно)

Размерность массива была объявлена в качестве другого ненулевого значения из его предыдущего объявления. Допускается повторное объявление размера массива, который ранее был объявлен с нулевым измерением, но не иначе.

array index out of bounds *(Синтактический анализатор)*

(индекс массива вне границ)

Массив индексируется с постоянным значением, которое меньше нуля или больше или равно количеству элементов в массиве.

assertion *(Генератор кода)*

(утверждение)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

assertion failed: \* *(Компоновщик)*

(ошибка утверждения: \*)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

attempt to modify const object *(Синтактический анализатор)*

(попытка изменить константный объект)

Объекты, объявленные "const", не могут быть присвоены или изменены каким-либо другим способом.

auto variable \* should not be qualified *(Синтактический анализатор)*

(автоматическая переменная \* не может быть квалифицирована)

Автоматическая переменная не должна иметь связанных с ней квалификаторов таких как "near" или "far". Ее класс памяти неявно определяется организацией стека.

bad #if ... defined() syntax *(Препроцессор)*

(неверный синтаксис #if... defined())

Псевдо-функция defined() в выражении препроцессора требует, чтобы ее аргумент был единственным именем. Имя должно начинаться с буквы. Он должен быть заключен в круглые скобки.

bad '-p' format *(Компоновщик)*

(неверный формат '-p')

Опция "-P", предоставленная компоновщику, неправильно сформирована.

bad -a spec: \* *(Компоновщик)*

(недопустимая спецификация –a: \*)

Формат спецификации -A, задающая диапазоны адресов компоновщику, имеет недопустимый формат. Корректный формат:

-Aclass=low-high

где class - имя класса psect, и low и high являются шестнадцатеричными числами.

bad -m option: \* *(Генератор кода)*

(недопустимая спецификация –m: \*)

Генератору кода была передана опция -M, которую он не понимает. Это не должно происходить, если он вызывается стандартным драйвером компилятора.

bad -q option \* *(Синтактический анализатор)*

(недопустимая спецификация –q \*)

Первый проход компилятора был вызван с опцией -Q, определяющей имя квалификатора типа, которое плохо сформировано.

bad arg \* to tysize *(Синтактический анализатор)*

(недопустимый аргумент tysize)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad arg to im *(Ассемблер)*

(недопустимый аргумент im)

Код операции "IM" в качестве аргумента принимает только константы 0, 1 или 2.

bad bconfloat - \* *(Генератор кода)*

(недопустимый bconfloat - \*)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad bit number *(Ассемблер, Оптимизатор)*

(недопустимый номер бита)

Номер бита должен быть абсолютным выражением в диапазоне 0-7.

bad bitfield type *(Синтактический анализатор)*

(недопустимый тип битового поля)

Битовое поле может иметь тип int.

bad character const *(Синтаксический анализатор, Ассемблер, Оптимизатор)*

(недопустимая символьная константа)

Эта символьная константа плохо сформирована.

bad character in extended tekhex line \* *(Objtohex)*

(некорректный символ в расширенной строке tekhex)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad checksum specification *(Компоновщик)*

(Неверная спецификация контрольной суммы)

Список контрольной суммы, предоставленный компоновщику, синтаксически неверен.

bad combination of flags *(Objtohex)*

(неправильная комбинация флагов)

Недопустимая комбинация опций, предоставленных для objtohex.

bad complex range check *(Компоновщик)*

(неверный комплексный контроль диапазона)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad complex relocation *(Компоновщик)*

(недопустимое сложное перемещение)

Компоновщику было предложено выполнить сложное перемещение, которое не является синтаксически правильным. Вероятно, это означает поврежденный объектный файл.

bad confloat - \* *(Генератор кода)*

(недопустимый confloat)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad conval - \* *(Генератор кода)*

(недопустимый conval)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad dimensions *(Генератор кода)*

(недопустимая размерность)

Генератору кода было передано объявление, которое приводит к тому, что массив имеет нулевой размер.

bad dp/nargs in openpar: c = \* *(Препроцессор)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad element count expr *(Генератор кода)*

(неверное количество элементов выражения)

В промежуточном коде есть ошибка. Попробуйте переустановить компилятор из дистрибутивных дисков, поскольку это может быть вызвано поврежденным файлом.

bad gn *(Генератор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad high address in -a spec *(Компоновщик)*

(неправильный старший адрес в спецификации -a)

Старший адрес, указанный в спецификации -A, недействителен: он должен быть допустимым числом, в десятичной, восьмеричной или шестнадцатеричной системе счисления. Система счисления задается конечным символом O (для восьмеричного) или H для шестнадцатеричного значения. По умолчанию значение десятичное.

bad int. code *(Генератор кода)*

(Некорректный внешний код)

Входная информация поступившая в генератор кода не является синтаксически правильной.

bad load address in -a spec *(Компоновщик)*

(неправильный адрес загрузки в спецификации -a)

Адрес загрузки, указанный в спецификации -A, недействителен: он должен быть допустимым числом, в десятичной, восьмеричной или шестнадцатеричной системе счисления. Система счисления задается конечным символом O (для восьмеричного

bad low address in -a spec *(Компоновщик)*

(неправильный младший адрес в спецификации -a)

Младший адрес, указанный в спецификации -A, недействителен: он должен быть допустимым числом, в десятичной, восьмеричной или шестнадцатеричной системе счисления. Система счисления задается конечным символом O (для восьмеричного) или H для шестнадцатеричного значения. По умолчанию значение десятичное.

bad min (+) format in spec *(Компоновщик)*

Минимальная спецификация минимального адреса в параметре компоновщика -p имеет неправильный формат.

bad mod '+' for how = \* *(Генератор кода)*

Внутренняя ошибка. Свяжитесь с HI-TECH.

bad non-zero node in call graph *(Компоновщик)*

Компоновщик обнаружил узел верхнего уровня в графе вызовов, на который ссылается нижний уровень в графе вызовов. Вероятно, это означает, что программа имеет косвенную рекурсию, которая не допускается при использовании скомпилированного стека.

bad object code format *(Компоновщик)*

(недопустимый формат объектного кода)

Недопустимый формат кода в этом объектном файле. Это, вероятно, означает, что он или усечен, поврежден, или не является объектным файлом HI-TECH.

bad op \* to revlog *(Генератор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad op \* to swaplog *(Генератор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad op: "\*" *(Генератор кода)*

Это вызвано ошибкой кода в промежуточном файле. Возможно, нет места на диске для временных файлов.

bad operand *(Оптимизатор)*

(недопустимый операнд)

Недопустимый операнд. Проверьте синтаксис.

bad origin format in spec *(Компоновщик)*

(недопустимый формат источника в спецификации)

Исходный формат опции -p не является допустимым десятичным, восьмеричным или шестнадцатеричным числом. Шестнадцатеричное число должно иметь заключительный символ H.

bad overrun address in -a spec *(Компоновщик)*

(недопустимый адрес в спецификации -a)

Адрес переполнения, заданный в спецификации -A, является недопустимым: он должен быть допустимым числом, в десятичной, восьмеричной или шестнадцатеричной системе счисления. Система счисления задается конечным символом O (для восьмеричного) или H для шестнадцатеричного значения. По умолчанию значение десятичное.

bad pragma \* *(Генератор кода)*

(недопустимая pragma)

Генератору кода была передана директива "pragma", которую он не понимает.

bad record type \* *(Компоновщик)*

(недопустимый тип записи)

Это указывает на то, что файл не является допустимым объектный файлом HI-TECH.

bad relocation type *(Ассемблер)*

(недопустимый тип перемещения)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad repeat count in -a spec *(Компоновщик)*

(неверное число повторений в спецификации -a)

Число повторений, указанное в спецификации -A, недействительно: оно должно быть допустимым десятичным числом.

bad ret\_mask *(Генератор кода)*

(недопустимая маска возврата)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad segment fixups *(Objtohex)*

(неверные исправления сегментов)

Это неявное сообщение от objtohex, которое вряд ли произойдет на практике.

bad segspec \* *(Компоновщик)*

(недопустимая спецификация сегмента)

Спецификация сегмента опции компоновщика (-G) недопустима. Правильная форма спецификации сегмента находится в следующих строках:

-Gnxc+o

где n соответствует номеру сегмента, x — символ множителя, c — константа (множитель), а o — постоянное смещение. Например, параметр

-Gnx4+16

присвоит селекторы сегмента, начиная с 16 и увеличивающиеся на 4 для каждого сегмента, т.е. в порядке 16, 20, 24 и т.д.

bad size in -s option *(Компоновщик)*

(недопустимый размер в опции -s)

Поле размера в опции –S имеет некорректный формат . Число должно быть в десятичной, восьмеричной или шестнадцатеричной системе счисления. Шестнадцатеричное число должно завершаться символом H и восьмеричное число - символом O. Предполагается, что все остальные являются десятичными.

bad size in index\_type *(Синтактический анализатор)*

(недопустимый размер index\_type)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad size list *(Синтактический анализатор)*

(недопустимый размер списка)

Первый проход компилятора был вызван с опцией -Z с указанием размеров типов, которые плохо сформированы.

bad storage class *(Генратор кода)*

(плохой класс памяти)

Класс памяти "auto" может использоваться только внутри функции. Параметры функции могут не иметь спецификатора класса памяти, кроме "register". Если эта ошибка выдается генератором кода, это может означать, что промежуточный файл кода является недопустимым. Она может быть вызвана нехваткой места на диске.

bad string \* in psect pragma *(Генратор кода)*

(Неверная строка в pragma psect)

Генератору кода была передана директива "pragma psect", имеющая неправильный формат строки. После "pragma psect" должна следовать последовать в форме "oldname=newname".

bad sx *(Генратор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad u usage *(Генратор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

bad variable syntax *(Генратор кода)*

(Недопустимый синтаксис переменной)

В промежуточном файле кода есть ошибка. Это может быть вызвано нехваткой дискового пространства для временных файлов.

bad which \* after i *(Генератор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

binary digit expected *(Синтаксический анализатор)*

(ожидается двоичная цифра)

Ожидалось двоичное число. Формат двоичного числа - 0Bxxx, где xxx - это строка, содержащая нули и / или единицы, например

0B0110

bit field too large (\* bits) *(Генератор кода)*

(битовое поле слишком длинное)

Максимальное число битов в битовом поле совпадает с числом битов в "int".

bit range check failed \* *(Компоновщик)*

(Сбой проверки битового диапазона)

Адресация битов выходит за пределы допустимого диапазона.

bitfield comparison out of range *(Генератор кода)*

(сравнение битового поля вне диапазона)

Это - результат сравнения битового поля со значением, когда значение битового поля вне диапазона. Например, сравнение 2-разрядного битового поля со значением 5 никогда не будет истиной, поскольку 2-разрядное битовое поле имеет диапазон от 0 до 3.

bug: illegal \_\_macro \* *(Препроцессор)*

(ошибка: недопустимый \_\_macro \*)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

call depth exceeded by \* *(Компоновщик)*

(превышена глубина вызова \*)

В графе вызовов показано, что функции вложены в глубину, превышающую указанную.

can’t allocate memory for arguments *(Препроцессор, Синтаксический анализатор)*

(не удается выделить память для аргументов)

Компилятору не удалось выделить больше памяти. Попробуйте увеличить размер доступной памяти.

can’t allocate space for port variables: \* *(Генератор кода)*

(не удается выделить память для переменных port)

Переменные "port" могут только быть объявлены "extern" или иметь абсолютный адрес, связанный через конструкцию "@ address". Они не могут быть объявлены таким образом, который потребовал бы, чтобы компилятор выделил место для них.

can’t be both far and near *(Синтаксический анализатор)*

(не может быть как far, так и near)

Недопустимо одновременно квалифицировать тип как far так и near.

can’t be long *(Синтаксический анализатор)*

(не может быть long)

Только "int" и "float" могут быть квалифицированы с помощью "long". Таким образом комбинации подобные "long char" недопустимы.

can’t be register *(Синтаксический анализатор)*

(не может быть register)

Только параметры функции или автоматические (локальные) переменные могут быть объявлены "register".

can’t be short *(Синтаксический анализатор)*

(не может быть short)

Только "int" могут быть квалифицированы с помощью "short". Таким образом комбинации подобные "short float" недопустимы.

can’t be unsigned *(Синтаксический анализатор)*

(не может быть unsigned)

Нет такого понятия, как число без знака с плавающей запятой.

can’t call an interrupt function *(Синтаксический анализатор)*

(вызов функции обработки прерываний недопустим)

Функция квалифицированная "interrupt" не может быть вызвана из других функций. Ее могут вызвать только аппаратные средства (или программное обеспечение)с помощью прерывания. Это вызвано тем, функции обработки прерываний имеет специальный функциональный код входа и выхода, который является подходящим только для вызова из прерывания. Функция "interrupt" может вызвать другие функции не предназначенные для обработки прерываний.

can’t create \* *(Генератор кода, Ассеблер, Компоновщик, Оптимизатор)*

(не удается создать \*)

Названный файл не может быть создан. Проверьте, что все каталоги по пути присутствуют.

can’t create cross reference file \* *(Ассемблер)*

(не удается создать файл перекрестных ссылок \*)

Не удалось создать файл перекрестных ссылок. Проверьте, что все каталоги присутствуют. Это может также быть вызвано нехваткой памяти для выполнения ассемблера.

can’t create temp file *(Компоновщик)*

(не удается создать временный файл)

Компилятор не смог создать временный файл. Проверьте переменную TEMP (и TMP) окружения DOS и убедитесь, что она указывает на существующий каталог, и что на этом диске есть свободное место. Например, AUTOEXEC.BAT должен содержать что-то вроде:

SET TEMP=C:\TEMP

где каталог C:\TEMP существует.

can’t create temp file \* *(Генератор кода)*

(не удается создать временный файл \*)

Компилятор не смог создать временный названный файл. Проверьте существование всех каталогов в пути к файлу.

can’t enter abs psect *(Ассемблер)*

(не удается ввести абсолютную psect)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

can’t find op *(Ассемблер,опримизатор)*

(не удается найти op)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

can’t find space for psect \* in segment \* *(Компоновщик)*

(не удается найти место для psect \* в сегменте \*)

Названная psect не может быть помещена в указанный сегмент. Этот означает, что память, связанная с сегментом, была заполнена, или что psect не может быть расположена ни в один из доступных промежутков в памяти. Разделите большие функции (для сегментов кода) на несколько меньших функций и убедитесь, что используются оптимизаторы.

can’t generate code for this expression *(Генератор кода)*

(не удается создать код для этого выражения)

Это выражение слишком сложно для обработки генератором кода. Попробуйте упростить выражение, например, используя временную переменную для хранения промежуточного результата.

can’t have 'port' variable: \* *(Генератор кода)*

(нельзя иметь переменную 'port': \*)

Квалификатор "port" может использоваться только с указателями или абсолютными переменными. Вы не можете определить переменную port, поскольку компилятор не выделяет место для переменных port. Вы можете объявить внешнюю переменную порта внешней (external).

can’t have 'signed' and 'unsigned' together *(Синтаксический анализатор)*

('signed' и 'unsigned' не могут быть вместе)

Модификаторы типа signed и unsigned не могут использоваться вместе в том же объявлении, так как они имеют противоположное значение.

can’t have an array of bits or a pointer to bit *(Синтаксический анализатор)*

(не может быть массива битов или указателя на бит)

Незаконно иметь массив битов или указатель на бит.

can’t have array of functions *(Синтаксический анализатор)*

(не может быть массива функций)

Невозможно получить массив функций. Однако можно иметь массив указателей на функции. Корректный синтаксис для массива указателей на функции - "int (\* arrayname[])();". Обратите внимание, что круглые скобки используются для связывания звездочки (\*) с именем массива перед круглыми скобками, обозначающими функцию.

can’t initialize arg *(Синтаксический анализатор)*

(не удается инициализировать аргумент)

Аргумент функции не может иметь инициализатор. Инициализация аргумента происходит, когда функция вызывается, и значение предоставляется аргументу вызывающей функцией.

can’t mix proto and non-proto args *(Синтаксический анализатор)*

(нельзя смешивать прототипированные и непрототипированные аргументы)

Объявление функции может содержать только все прототипированные аргументы (т.е. с типами внутри круглых скобок) или все аргументы в стиле K&R (т.е. только имена внутри круглых скобок и типы аргументов в списке объявлений до начала тела функции).

can’t open *(Компоновщик)*

(не удается открыть)

Файл не может быть открыт - проверьте правописание.

can’t open \* *(Генератор кода, Ассеблер, Оптимизатор, Cromwell)*

(не удается открыть \*)

Не удалось открыть названный файл. Проверьте правильность написания и пути к каталогу. Это также может быть вызвано нехваткой памяти.

can't open avmap file \* *(Компоновщик)*

(не удается открыть файл avmap \*)

Файл, требуемый для создания файлов символов формата Avocet, отсутствует. Попытайтесь переустановить компилятор.

can't open checksum file \* *(Компоновщик)*

(не удается открыть файл контрольной суммы \*)

Не удается открыть файл контрольной суммы, определенный в objtohex. Проверьте правописание и т.д.

can't open command file \* *(Препроцессор,Компоновщик)*

(не удается открыть командный файл \*)

Определенный командный файл не удалось открыть для чтения. Проверьте правописание!

can't open error file \* *(Компоновщик)*

(не удается открыть файл ошибок \*)

Не удалось открыть файл ошибок, указанный с помощью опции -e.

can't open include file \* *(Ассемблер)*

(не удается открыть включаемый файл)

Не удалось открыть названный включаемый файл. Проверьте правописание. Она также может быть вызвана исчерпанием памяти или исчерпанием дескрипторов файлов.

can't open input file \* *(Препроцессор, Ассемблер)*

(не удается открыть входной файл)

Не удалось открыть указанный входной файл. Проверьте написание имени файла.

can't open output file \* *(Препроцессор, Ассемблер)*

(не удается открыть выходной файл)

Не удалось создать указанный выходной файл. Возможно, каталог в имени пути не существует.

can't reopen \* *(Синтаксический анализатор)*

(не удается повторное открытие \*)

Компилятору не удалось повторно открыть только что созданный временный файл.

can't seek in \* *(Компоновщик)*

(не удается найти \*)

Компоновщик не может выполнить поиск в указанном файле. Удостоверьтесь, что выходной файл имеет допустимое имя файла.

can't take address of register variable *(Синтаксический анализатор)*

(не может принимать адрес регистровой переменной)

Переменная объявленная "register" не имеет места выделенного в памяти, и таким образом недопустимо, пытаться получить ее адрес, применяя оператор "&".

can't take sizeof func *(Синтаксический анализатор)*

(не удается получить размер функции)

Функции не имеют размеров, поэтому вы не можете использовать оператор "sizeof" для функции.

can't take sizeof (bit) *(Синтаксический анализатор)*

(не удается получить размер битов)

Вы не можете получить sizeof битового значения, так как он меньше байта.

can't take this address *(Синтаксический анализатор)*

(не удается получить этот адрес)

Выражение, которое было объектом оператора "&", не является тем, которое обозначает память ("значение lvalue"), и поэтому его адрес не может быть определен.

can't use a string in an #if *(Препроцессор)*

(недопустимо использовать строку в #if)

Препроцессор не допускает использования строк в выражениях #if.

cannot get memory *(Компоновщик)*

(не может получить память)

Компоновщику не хватает памяти! Это вряд ли произойдет, но удаление TSR и т.д. является средством исправления.

cannot open *(Компоновщик)*

(не удается открыть)

Файл не может быть открыт – проверьте правописание.

cannot open include file \* *(Препроцессор)*

(не удается открыть включаемый файл \*)

Названный включаемый файл не может быть открыт для чтения препроцессором. Проверьте правильность написания имени файла. Если это стандартный файл заголовка, а не в текущем каталоге, то имя должно быть заключено в угловые скобки (< >), а не кавычки.

cast type must be scalar or void *(Синтаксический анализатор)*

(тип приведения должен быть скалярным или void)

Приведенный тип (декларатор абстрактного типа, заключенный в круглые скобки) должен относиться к типу, который является либо скалярным (т. е. не массивом или структурой), либо типом "void".

char const too long *(Синтаксический анализатор)*

(символьная константа слишком длинная)

Символьная константа, заключенная в одинарные кавычки, может содержать не более одного символа.

character not valid at this point in format specifier *(Синтаксический анализатор)*

(символ, не допустимый в этой точке в спецификаторе формата)

Спецификатор формата функции printf() имеет недопустимый символ.

checksum error in intel hex file \*, line \* *(Cromwell)*

(Ошибка контрольной суммы в Intel hex \*, строке \*)

Ошибка контрольной суммы была найдена в указанной строке в указанном Intel hex файле. Файл, возможно, был поврежден.

circular indirect definition of symbol \* *(Компоновщик)*

(круговое косвенное определение символа \*)

Указанный символ приравнивается к внешнему символу, который, в свою очередь, был приравнен к первому символу.

class \* memory space redefined: \*/\* *(Компоновщик)*

(переопределение пространства памяти класса \*)

Класс был определен в двух различных пространствах памяти. Переименуйте один из классов или, если они являются одним и тем же классом, поместите их в одну и ту же область памяти.

close error (disk space?) *(Синтаксический анализатор)*

(ошибка закрытия (дисковое пространство?))

При закрытии компилятором временного файла возникла ошибка. Наиболее вероятной причиной этого является нехватка места на диске для файла.

common symbol psect conflict: \* *(Компоновщик)*

(конфликт общих символов psect: \*)

Общий символ определен в нескольких psect.

complex relocation not supported for -r or -I options yet *(Компоновщик)*

(сложное перемещение не поддерживается для опций -r или -I)

Компоновщику была предоставлена опция -R или -L с файлом, содержащим сложное перемещение. Это пока не поддерживается.

conflicting fnconf records *(Компоновщик)*

(конфликт записей fnconf)

Возможно, это вызвано несколькими модулями startoff времени выполнения. Проверьте аргументы компоновщика или "Object Files..." (Объектные файлы...) в HPD.

constant conditional branch *(Генератор кода)*

(Постоянная условная ветвь)

Условная ветвь (созданная с помощью оператора "if" и т. д.) всегда соответствует одному и тому же пути. Это может означать выражение с отсутствующими или некорректными скобками, что приводит к тому, что вычисление дает значение, отличающееся от ожидаемого, или это может быть вызвано тем, что вы написали нечто вроде "while(1)". Для создания бесконечного цикла используйте "for(;;)".

constant conditional branch: possible use of = instead of == *(Генератор кода)*

(постоянная условная ветвь: возможно использование = вместо ==)

Внутри if или другой условной конструкции есть выражение, где константа присваивается переменной. Это может означать, что вы случайно использовали назначение (=) вместо сравнения (==).

constant expression required *(Синтаксический анализатор)*

(требуется константное выражение)

В этом контексте требуется выражение, которое может быть оценено к константе во время компиляции.

constant left operand to ? *(Генератор кода)*

(левый операнд ? константа)

Левый операнд условного оператора (?) является константой, таким образом, результат третичного оператора ?: всегда будет одинаковым.

constant operand to || or && *(Генератор кода)*

(константа является операндом || или &&)

Один операнд логического оператора || или && является константой. Проверьте выражение на отсутствие или неправильное размещение скобок.

constant relational expression *(Генератор кода)*

(постоянное реляционное выражение)

Существует реляционное выражение, которое всегда будет истинным или ложным. Например, сообщение может быть связано с тем, что вы сравниваете число без знака с отрицательным значением или сравниваете переменную со значением, большим, чем наибольшее число, которое она может представлять.

control line \* within macro expansion *(Препроцессор)*

(строка управления \* в макрорасширении)

При разворачивании макроса обнаружена управляющая строка препроцессора (начинающаяся с #). Этого не должно происходить.

declaration of \* hides outer declaration *(Синтаксический анализатор)*

(объявление \* скрывает внешнее описание)

Объявлен объект, который имеет то же имя, что и внешнее описание (т. е. одно внешнее и предшествующее текущей функции или блоку). Это допустимо, но может привести к случайному использованию переменной, когда подразумевалась внешняя.

declarator too complex *(Синтаксический анализатор)*

(оператор объявления слишком сложный)

Этот оператор объявления слишком сложен для обработки компилятором. Проанализируйте объявление и найдите способ упростить его. Если компилятор находит его слишком сложным, непонятно кто будет поддерживать код.

default case redefined *(Синтаксический анализатор)*

(case по умолчанию переопределен)

В инструкции switch допускается только один оператор case "default". У вас их несколько.

deff not supported in cp/m version *(Ассемблер)*

(deff не поддерживается в cp/m версии)

Ассемблер CP/M не поддерживает плавающую точку.

def[bmsf] in text psect *(Оптимизатор)*

(def[bmsf] в psect text)

Недопустимый файл ассемблера, предоставленный оптимизатору.

degenerate signed comparison *(Генератор кода)*

(вырожденное знаковое сравнение)

Существует сравнение значения со знаком с наиболее отрицательной величиной, возможной для этого типа, такого, что сравнение всегда будет истиной или ложью. Например, char c;

if(c>=-128)

всегда будет истина (true), поскольку 8-разрядный символ со знаком имеет максимальное отрицательное значение -128.

degenerate unsigned comparison *(Генератор кода)*

(вырожденное беззнаковое сравнение)

Существует сравнение беззнакового значения с нулевым значением, которое всегда будет истиной или ложью. Например

unsigned char c;

if(c >= 0)

всегда будет истина (true), поскольку беззнаковое значение никогда не может быть меньше нуля.

delete what ? *(Библиотекарь)*

(удалить что?)

Библиотекарь требует, чтобы один или несколько модулей были перечислены для удаления при использовании ключа 'd'.

did not recognize format of input file *(Cromwell)*

(не распознает формат входного файла)

Входной файл для Cromwell должен быть в формате COD, Intel HEX, Motorola HEX, COFF, OMF51, P&E или HI-TECH

digit out of range (Анализатор, Ассемблер, Оптимизатор)

(цифра вне диапазона)

Цифра в этом числе выходит за пределы диапазона системы счисления для числа, например, использования цифры 8 в восьмеричном числе или шестнадцатеричных цифр A-F в десятичном числе. Восьмеричное число обозначается строкой цифр, начинающейся с нуля, в то время как шестнадцатеричное число начинается с "0X" или "0x".

dimension required *(Синтаксический анализатор)*

(требуется размерность)

Только старшему (т. е. первому) измерению в многомерном массиве может быть не присвоено значение. Все последующие измерения должны быть присутствовать.

direct range check failed \* *(Компоновщик)*

(не удалось выполнить проверку прямого диапазона \*)

Прямая адресация была вне допустимого диапазона.

directive not recognized *(Ассемблер)*

(директива не распознана)

Ассемблерная директива не распознана. Проверьте правописание.

divide by zero in #if, zero result assumed *(Препроцессор)*

(деление на ноль в #if, предполагается нулевой результат)

Внутри выражения #if существует деление на ноль, которое обрабатывается как нулевое значение.

division by zero *(Генератор кода)*

(деление на ноль)

Вычисленное константное выражение включало деление на ноль..

double float argument required *(Синтаксический анализатор)*

(требуется аргумент с плавающей запятой двойной точности)

Спецификатор формата printf, соответствующий этому параметру, является %f или подобный, и требует выражения с плавающей точкой. Проверьте наличие отсутствующих или дополнительных спецификаторов формата или параметров в printf.

duplicate -d or -h flag *(Компоновщик)*

(повторение флага -d или -h)

Имя файла символов определено компоновщику повторно.

duplicate -m flag *(Компоновщик)*

(повторение флага -m)

Линкеру нравится только один флаг -m, если только один из них не указывает имя файла. Два имени map файла больше, чем он может обработать!

duplicate case label *(Генератор кода)*

(дублирование метки case)

В этом операторе switch имеются две метки case с одинаковым значением.

duplicate label \* *(Синтаксический анализатор)*

(повторяющаяся метка \*)

Одно и то же имя используется для метки более одного раза в этой функции. Обратите внимание, что область действия метки - вся функция, а не только блок, содержащий метку.

duplicate qualifier *(Синтаксический анализатор)*

(повторяющийся квалификатор)

В данной спецификации типа имеется два вхождения одного и того же квалификатора. Это может произойти либо непосредственно, либо с помощью оператора typedef. Удалите избыточный квалификатор.

duplicate qualifier key \* *(Синтаксический анализатор)*

(повторяющийся ключ квалификатора)

Этот ключ спецификатора (заданный с помощью опции -Q) использовался дважды.

duplicate qualifier name \* *(Синтаксический анализатор)*

(повторяющееся имя квалификатора)

С помощью опции -Q первому проходу было определено повторяющееся имя квалификатора. Этого не должно происходить, если используются стандартные драйверы компилятора.

end of file within macro argument from line \* *(Препроцессор)*

(конец файла в аргументе макроса из строки \*)

Параметр макроса не был завершен. Это, вероятно, означает, что в макро-вызове была опущена закрывающая скобка. Приведенный номер строки обозначает строку, в которой начался аргумент макроса.

end of string in format specifier *(Синтаксический анализатор)*

(конец строки в спецификаторе формата)

Описатель формата для функции стиля printf() имеет неправильный формат.

end statement inside include file or macro *(Ассемблер)*

(конец оператор внутри включает файл или макроса)

Был найден оператор END во включаемом файле или макросе.

entry point multiply defined *(Компоновщик)*

(многократное определение точки входа)

В объектных файлах имеется более одной точки входа, заданной компоновщиком.

enum tag or { expected *(Синтаксический анализатор)*

(ожидается тег enum или {)

После ключевого слова "enum" должен быть либо идентификатор, который является или будет определен как тег перечисления, либо открывающая фигурная скобка.

eof in #asm *(Препроцессор)*

(eof в #asm)

В блоке #asm обнаружен конец файла. Вероятно, это означает, что #endasm отсутствует или с ошибкой.

eof in comment *(Препроцессор)*

(eof в комментарии)

В комментарии обнаружен конец файла. Проверьте отсутствующий флаг закрывающий комментарий.

eof inside conditional *(Ассемблер)*

(eof в условном выражении)

При сканировании "endif" был обнаружен конец файла, совпадающий с предыдущим "if".

eof inside macro def'n *(Ассемблер)*

(eof в макро-определении)

При обработке определения макроса был обнаружен конец файла. Это означает, что отсутствует директива "endm".

eof on string file *(Синтаксический анализатор)*

(eof в строковом файле)

Первый проход обнаружил неожиданный конец файла при повторном чтении файла, используемого для хранения постоянных строк перед сортировкой и слиянием. Скорее всего, это связано с нехваткой места на диске. Проверьте свободное место на диске.

error closing output file *(Генератор кода, Оптимизатор)*

(ошибка закрытия выходного файла)

Компилятор обнаружил ошибку при закрытии файла. Это, скорее всего, означает, что на диске недостаточно места.

error dumping \* *(Cromwell)*

(ошибочный дамп \*)

Или входной файл Cromwell не поддерживается, или дамп этого файла не может быть выведен на экран.

error in format string *(Синтаксический анализатор)*

(ошибка в строке формата)

Ошибка в строке формата. Строка интерпретируется как строка формата стиля printf(), и она не является синтаксически правильной. Если не исправить, это приведет к неожиданному поведению во время выполнения.

evaluation period has expired *(Драйвер)*

(период оценки истек)

Срок действия этого компилятора истек. Свяжитесь с HI-TECH для приобретения полной лицензии.

expand - bad how *(Генератор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

expand - bad which *(Генератор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

expected '-' in -a spec *(Компоновщик)*

(в спецификации -a ожидается '-')

Спецификация -A между старшим и младшим адресами должна содержать знак минус (-), например

-AROM=1000h-1FFFh

exponent expected *(Синтаксический анализатор)*

(ожидается экспонента)

Константа с плавающей запятой после "e" или "E" должна иметь по крайней мере одну цифру.

expression error *(Генератор кода, Ассемблер, Оптимизатор)*

(ошибка выражения)

В этом выражении существует синтаксическая ошибка или ошибка в промежуточном файле кода. Это может быть вызвано переполнением места на диске.

expression generates no code *(Генератор кода)*

(выражение не создает код)

Это выражение не создает кода. Проверьте наличие таких вещей, как отсутствие скобок в вызове функции.

expression stack overflow at op \* *(Препроцессор)*

(переполнение стека выражений в операции \*)

Выражения в строках #if оцениваются с помощью стека размером 128. Очень сложные выражения могут вызвать его переполнение. Упростите выражение.

expression syntax *(Синтаксический анализатор)*

(Синтаксис выражения)

Это выражение плохо сформировано и не может быть проанализировано компилятором.

expression too complex *(Синтаксический анализатор)*

(слишком сложное выражение)

Это выражение вызвало переполнение внутреннего стека компилятора и должно быть перестроено или разделено на два выражения.

external declaration inside function *(Синтаксический анализатор)*

(внешнее объявление внутри функции)

Функция содержит объявление "extern". Это допустимо, но неизменно плохо, так как оно ограничивает область объявления функции телом функции. Это означает, что если компилятор обнаруживает другое объявление, использование или определение внешнего объекта позже в том же файле, то он больше не будет иметь более раннее объявление и таким образом не сможет проверить согласованность объявлений. Это может привести к необычному поведению программы или ошибкам подписи во время компоновки. Оно также скрывает любые предыдущие объявления одного и того же, снова подменяя проверку типов компилятором. Как правило всегда объявляйте переменные "extern" и функции вне любых других функций.

fast interrupt can't be used in large model *(Генератор кода)*

(быстрое прерывание не может использоваться в большой модели)

Большая модель (с переключением банков) не поддерживает быстрые прерывания, поскольку альтернативный набор регистров используется для коммутации банков.

field width not valid at this point *(Синтаксический анализатор)*

(недопустимая ширина поля в этой точке)

Ширина поля не может отображаться в этой точке спецификатора формата printf().

file name index out of range in line no. record *(Cromwell)*

(индекс имени файла вне диапазона в строке № записи)

Файл .COD имеет недопустимый формат в указанной записи.

filename work buffer overflow *(Препроцессор)*

(переполнение рабочего буфера имени файла)

Имя файла, созданное при поиске включаемого файла, превысило длину внутреннего буфера. Поскольку этот буфер содержит 4096 байт, это вряд ли произойдет.

fixup overflow in expression \* *(Компоновщик)*

(исправление переполнения в выражении \*)

Компоновщику было предложено переместить (исправить) элемент, который не поместился бы обратно в пространство после переноса. Например, это произойдет, если объект размера байта будет инициализирован адресом, который больше 255. Эта ошибка возникла в сложном выражении.

fixup overflow referencing \* *(Компоновщик)*

(исправление переполнения ссылки \*)

Компоновщику предложено переместить (исправить) элемент, который не впишется назад в пространство после перемещения. Например, это произойдет, если объект размера байта инициализирован адресом, который больше 255.

flag \* unknown *(Ассемблер)*

(флаг \* не опознан)

Эта опция, используемая на директиве "PSECT", неизвестна ассемблеру.

float param coerced to double *(Синтаксический анализатор)*

(параметр float приведен к double)

Если функция без прототипа имеет параметр, объявленный как "float", компилятор преобразует его в "double float". Это связано с тем, что соглашения о преобразования типов C по умолчанию предусматривают, что когда число с плавающей запятой передается не прототипированной функции, оно преобразуется в double. Важно, чтобы объявление функции соответствовало этому соглашению.

floating exponent too large *(Ассемблер)*

(Показатель числа с плавающей запятой слишком велик)

Показатель числа с плавающей запятой слишком велик. Для Z80 наибольший показатель с плавающей запятой равен десятичному значению 19.

floating number expected *(Ассемблер)*

(ожидается число с плавающей запятой)

Параметрами псевдо оператора "DEFF" должны быть допустимые числа с плавающей запятой.

formal parameter expected after # *(Препроцессор)*

(после # ожидается формальный параметр)

За оператором превращения в строку # (не путать с ведущим #, используемым для строк управления препроцессором) должен следовать формальный макро-параметр. Если вам требуется выровнять токен, вам нужно определить специальный макрос, чтобы сделать это, например.

#define \_\_mkstr\_\_(x) #x

тогда используйте \_\_mkstr\_\_ (токен) везде, где необходимо преобразовать токен в строку.

function \* appears in multiple call graphs: rooted at \* *(Компоновщик)*

(функция \* присутствует в нескольких графах вызовов: с корнем \*)

Эта функция может быть вызвана как из основного линейного кода так и из кода прерывания. Используйте ключевое слово reentrant, если этот компилятор поддерживает его, или перекодируйте, чтобы избежать использования локальных переменных или параметров или дублируйте функцию.

function \* argument evaluation overlapped *(Компоновщик)*

(Определение аргумента функции \* перекрывается)

Вызов функции включает аргументы, которые перекрываются между двумя функциями. Это может произойти при вызове типа:

void fn1(void) {

fn3( 7, fn2(3), fn2(9)); /\* Проблемный вызов \*/

}

char fn2( char fred) {

return fred + fn3(5,l,0);

}

char fn3(char one, char two, char three) {

return one+two+three;

}

где fn1 вызывает fn3, а два аргумента вычисляются путем вызова fn2, которая в свою очередь вызывает fn3. Чтобы предотвратить это, необходимо изменить структуру.

function \* is never called *(Компоновщик)*

(функция \* никогда не вызывается)

Эта функция никогда не вызывается. Это может не представлять проблем, но можно сохранить пространство, удалив ее. Если вы считаете, что эту функцию нужно вызвать, проверьте исходный код.

function body expected *(Синтаксический анализатор)*

(ожидается тело функции)

В случае, когда объявление функции встречается с аргументами стиля K&R (т. е. именами аргументов, но без типов в круглых скобках), ожидается, что будет следовать тело функции.

function declared implicit int *(Синтаксический анализатор)*

(функция неявно объявлена int)

Если компилятор обнаруживает вызов функции, имя которой в настоящее время не определено, компилятор автоматически объявляет функцию типа "int" с неопределенными параметрами (стиль K&R). Если впоследствии будет обнаружено определение функции, возможно, ее тип и аргументы будут отличаться от предыдущего неявного объявления, что приведет к ошибке компилятора. Решение заключается в обеспечении того, чтобы все функции были определены или, по крайней мере, объявлены перед использованием, желательно с прототипами параметров. Если необходимо сделать предварительное объявление функции, то перед ней должны предшествовать ключевые слова "extern" или "static".

function does not take arguments *(Анализатор, Генератор кода)*

(функция не принимает аргументов)

Эта функция не имеет параметров, но вызывается здесь с одним или несколькими аргументами.

function is already 'extern'; can't be 'static' *(Синтаксический анализатор)*

(функция уже - 'extern'; не может быть 'static')

Эта функция уже была объявлена extern, возможно через неявное объявление. Теперь она повторно объявлена static, но это повторное объявление недопустимо. Если проблема возникла из-за использования перед определением, либо переместите определение в файле выше, или поместите предшествующее определение static в файле раньше, например,

static int fred(void);

function or function pointer required *(Синтаксический анализатор)*

(требуется функция или указатель функции)

Объектом вызова функции может быть только указатель на функцию или функция. Эта ошибка может быть получена, когда в выражении присутствует синтаксическая ошибка, приводящая к переменной или выражению, сопровождаемому открывающейся круглой скобкой ("("), обозначающей вызов функции.

functions can't return arrays *(Синтаксический анализатор)*

(функции не могут возвращать массивы)

Функция может возвратить только скалярный (простой) тип или структуру. Она не может возвратить массив.

functions can't return functions *(Синтаксический анализатор)*

(функции не могут возвращать функции)

Функция не может возвратить функцию. Она может возвратить указатель функции. Функция, возвращая указатель на функцию может быть объявлена следующим образом: int (\* (name()))(). Обратите внимание на множество скобок, необходимых для правильной привязки частей объявления.

functions nested too deep *(Генератор кода)*

(функции вложены слишком глубоко)

Эта ошибка вряд ли возможна в коде C, так как C не может иметь вложенные функции!

garbage after operands *(Ассемблер)*

(неверная информация после операндов)

Есть что-то в этой строке после операндов, кроме комментария. Это может указывать на ошибку операнда.

garbage on end of line *(Ассемблер)*

(неверная информация в конце строки)

После окончания операндов для этой инструкции были непустые символы и символы без комментариев. Обратите внимание, что комментарий должен начинаться точкой с запятой.

hex digit expected *(Синтаксический анализатор)*

(ожидается шестнадцатеричная цифра)

После "0x" должна следовать по крайней мере одна из шестнадцатеричных цифр 0-9 и A-F или a-f.

I/O error reading symbol table *(Cromwell)*

(Ошибка ввода-вывода при чтении таблицы символов)

Утилита Cromwell не смогла прочитать таблицу символов. Это могло произойти потому, что файл был усечен или имелась какая-то другая проблема чтения файла.

ident records do not match *(Компоновщик)*

(идентификационные записи не совпадают)

Объектные файлы, переданные компоновщику, не имеют соответствующих идентификационных записей. Это означает, что они предназначены для разных типов процессоров.

identifier expected *(Синтаксический анализатор)*

(Ожидается идентификатор)

Внутри фигурных скобок объявления "enum" должен быть список идентификаторов, разделенных запятыми.

identifier redefined: \* *(Синтаксический анализатор)*

(идентификатор переопределен: \*)

Этот идентификатор уже определен. Он не может быть определен снова.

identifier redefined: \* (from line \*) *(Синтаксический анализатор)*

(идентификатор переопределен: \* (из строки \*))

Этот идентификатор был определен дважды. Значение 'from line' - это номер строки первого объявления.

illegal # command \* *(Препроцессор)*

(неверная команда \* после #)

Препроцессор обнаружил строку, начинающуюся с #, но за которой не следует распознанное ключевое слово управления. Это, вероятно, означает, что ключевое слово было неправильно написано. Допустимыми ключевыми словами управления являются: assert, asm, define, elif, else, endasm, endif, error, if, ifdef, ifndef, include, line, pragma, undef.

illegal #if line *(Препроцессор)*

(недопустимая строка #if)

В выражении после #if имеется синтаксическая ошибка. Проверьте выражение, чтобы убедиться, что оно правильно построено.

illegal #undef argument *(Препроцессор)*

(неверный аргумент #undef)

Аргумент #undef должен быть допустимым именем. Он должен начинаться с буквы.

illegal '#' directive *(Препроцессор, Синтаксический анализатор)*

(неверная директива '#')

Компилятор не понимает директиву "#". Возможно, это неправильно написание директивы препроцессора "#".

illegal character (\* decimal) in #if *(Препроцессор)*

(недопустимый символ (\* десятичный) в #if)

Выражение #if содержит недопустимый символ. Проверьте правильность синтаксиса строки.

illegal character \* *(Синтаксический анализатор)*

(недопустимый символ \*)

Этот символ недопустим.

illegal character \* in #if *(Препроцессор)*

(недопустимый символ \* в #if)

В выражении #if существует символ, который не должен в нем присутствовать. Допустимыми являются буквы, цифры и символы, состоящие из допустимых операторов.

illegal conversion *(Синтаксический анализатор)*

(недопустимое преобразование)

Это выражение подразумевает преобразование между несовместимыми типами, например, преобразование типа структуры в целое число.

illegal conversion between pointer types *(Синтаксический анализатор)*

(недопустимое преобразование между типами указателей)

Указатель одного типа (т.е. указывающий на конкретный тип объекта) был преобразован в указатель другого типа. Обычно это означает, что использовалась неправильная переменная, но если это действительно то, что нужно сделать, используйте приведение типа для того, чтобы уведомить компилятор о необходимости преобразования, и предупреждение будет подавлено.

illegal conversion of integer to pointer *(Синтаксический анализатор)*

(недопустимое преобразование целого числа в указатель)

Целое число присваивается или иным образом преобразуется в тип указателя. Обычно это означает, что вы использовали неправильную переменную, но если это действительно то, что вы хотите сделать, используйте приведение типов, чтобы сообщить компилятору, что вы хотите преобразовать, и предупреждение будет подавлено.

illegal conversion of pointer to integer *(Синтаксический анализатор)*

(недопустимое преобразование указателя в целое число)

Указатель был присвоен или иным образом преобразован в целочисленный тип. Обычно это означает, что вы использовали неправильную переменную, но если это действительно то, что вы хотите сделать, используйте приведение типов, чтобы сообщить компилятору, что вы хотите преобразовать, и предупреждение будет подавлено.

illegal flag \* *(Компоновщик)*

(недопустимый флаг \*)

Этот флаг не распознан.

illegal function qualifier(s) *(Синтаксический анализатор)*

(недопустимые квалификаторы функций)

К функции применен квалификатор, например "const" или "volatile". Эти квалификаторы имеют смысл только при использовании с lvalue (т. е. выражением, обозначающим хранение памяти). Возможно, вы не учли звездочку ("\*"), указывающую, что функция должна возвращать указатель на квалифицированный объект.

illegal initialisation *(Синтаксический анализатор)*

(недопустимая инициализация)

Вы не можете инициализировать объявление "typedef", потому что оно не резервирует место в памяти, которое может быть инициализировано.

illegal operation on a bit variable *(Синтаксический анализатор)*

(недопустимая операция с битовой переменной)

Поддерживаются не все операции с битовыми переменными. Эта операция является одной из них.

illegal operator in #if *(Препроцессор)*

(недопустимый оператор в #if)

Выражение #if содержит недопустимый оператор. Проверьте правильность синтаксиса.

illegal or too many -g flags *(Компоновщик)*

(недопустимый или много флагов -g)

Существует несколько опций, или опция -g не имеет аргументов, следующих за ним. Аргументы определяют, как рассчитываются адреса сегментов.

illegal or too many -o flags *(Компоновщик)*

(недопустимый или много флагов -o)

Этот флаг -o является незаконным или встречается другая опция -o. Опция компоновщика -o должна включать имя файла. Между именем файла и -o не должно быть пробела, например, -ofile.obj

illegal or too many -p flags *(Компоновщик)*

(недопустимый или много флагов -p)

Слишком много опций -p, передано компоновщику, или в опции -p отсутствуют параметры. Параметры отдельных опций -p могут быть объединены и разделены запятыми.

illegal record type *(Компоновщик)*

(недопустимый тип записи)

Ошибка в объектном файле. Это либо недопустимый объектный файл, либо внутренняя ошибка в компоновщике. Попробуйте воссоздать объектный файл.

illegal relocation size: \* *(Компоновщик)*

(недопустимый размер перемещения: \*)

Ошибка в формате объектного кода, считываемом компоновщиком. Это означает либо использование компоновщика, который устарел, либо наличие внутренней ошибки в ассемблере или компоновщике.

illegal relocation type: \* *(Компоновщик)*

(недопустимый тип перемещения: \*)

Объектный файл содержит запись перемещения с недопустимым типом перемещения. Вероятно, это означает, что файл поврежден или не является объектным файлом.

illegal switch \* *(Генератор кода, Ассемблер, Оптимизатор)*

(недопустимый switch \*)

Эта опция командной строки не была понята.

illegal type for array dimension *(Синтаксический анализатор)*

(недопустимый тип размерности массива)

Размерность массива должна быть или целочисленным типом или перечисляемым значением.

illegal type for index expression *(Синтаксический анализатор)*

(недопустимый тип индексного выражения)

Индексное выражение должно быть либо целым, либо перечислимым значением.

illegal type for switch expression *(Синтаксический анализатор)*

(недопустимый тип выражения switch)

Операция "switch" должна иметь выражение, являющееся либо целочисленным типом, либо перечислимым значением.

illegal use of void expression *(Синтаксический анализатор)*

(неверное использование выражения void)

Выражение void не имеет значения, поэтому его нельзя использовать везде, где требуется выражение со значением, например, в качестве операнда арифметического оператора.

image too big *(Objtohex)*

(образ слишком большой)

Образ программы, созданный objtohex, слишком велик для своей системы виртуальной памяти.

implicit conversion of float to integer *(Синтаксический анализатор)*

(неявное преобразование float в integer)

Значение с плавающей запятой было присвоено или иным образом преобразовано в целый тип. Это может привести к усечению значения с плавающей запятой. Приведенный тип приведет к тому, что это предупреждение исчезнет.

implicit return at end of non-void function *(Синтаксический анализатор)*

(неявный возврат в конце не-void функции)

Функция, объявленная для возврата значения, имеет путь выполнения, позволяющий ей дойти до конца тела функции, возвращаясь без значения. Либо вставьте оператор return со значением, либо если функция не возвращает значение, объявите ее "void".

implict signed to unsigned conversion *(Синтаксический анализатор)*

(неявное преобразование со знаком в без знака)

Число со знаком присваивается или иным образом преобразуется в больший беззнаковый тип. В соответствии с правилами ANSI "сохранение значения", это приведет к тому, что значение со знаком сначала будет расширено с помощью первого знака до числа со знаком, а затем преобразуется в беззнаковое (что не предполагает изменения битового шаблона). Таким образом может произойти непредвиденное расширение знака. Чтобы этого не произошло, необходимо сначала преобразовать значение со знаком в эквивалент без знака, например, если вы хотите присвоить символ со знаком к unsigned int, сначала преобразуйте значение типа char в "unsigned char".

inappropriate 'else' *(Синтаксический анализатор)*

(неуместное 'else')

Обнаружено ключевое слово "else", которое не может быть связано с оператором "if". Это может означать, что отсутствует фигурная скобка или имеется другая синтаксическая ошибка.

inappropriate break/continue *(Синтаксический анализатор)*

(неуместный break/continue)

Найден оператор "break" или "continue", который не заключен в соответствующую структуру управления, "continue" может использоваться только внутри цикла "while", "for" или "do while", тогда как "break" может используется только внутри этих циклов или в операторе "switch".

include files nested too deep *(Ассемблер)*

(включаемые файлы, вложены слишком глубоко)

Расширения макросов и включение обработки файлов заполнили внутренний стек ассемблера. Максимальное количество открытых макросов и включенных файлов - 30.

included file \* was converted to lower case *(Препроцессор)*

(включаемый файл \* был преобразован в нижний регистр)

Файл, указанный для включения, не найден, но вместо него был найден и использован файл с версией имени в нижнем регистре.

incompatible intermediate code version; should be \* *(Генератор кода)*

(несовместимая версия промежуточного кода; должно быть \*)

Промежуточный файл кода, произведенный P1, не является корректной версией для использования с этим генератором кода. Это или несовместимые версии одного или нескольких компиляторов были установлены в том же каталоге, или произошла ошибка временного файла, приведшая к его повреждению. Проверьте настройку переменной среды TEMP. Если она ссылается на длинное имя пути, измените его на нечто более короткое.

incomplete \* record body: length = \* *(Компоновщик)*

(неполное тело записи \*: длина = \*)

Объектный файл содержит запись с недопустимым размером. Вероятно, это означает, что файл усечен или не является объектным файлом.

incomplete ident record *(Библиотекарь)*

(неполная запись идентификатора)

Запись IDENT в объектном файле была неполной.

incomplete record *(Objtohex, Библиотекарь)*

(неполная запись)

Объектный файл, передаваемый в objtohex, или утилиту обработки библиотек поврежден.

incomplete record: \* *(Компоновщик)*

(неполная запись \*)

Запись объектного кода неполная. Вероятно, это происходит, из-за поврежденного или недопустимого объектного модуля. Перекомпилируйте исходный файл, наблюдая за ошибками исчерпания дискового пространства и т.п.

incomplete record: type = \* length = \*

(неполная запись: тип = \* длина = \*)

Это сообщение создается утилитами DUMP или XSTRIP и указывает, что объектный файл не является допустимым файлом объекта HI-TECH или что он был усечен, возможно, из-за нехватки места на диске или электронном диске.

incomplete symbol record *(Библиотекарь)*

(неполная запись символа)

Запись SYM в объектном файле была неполной.

inconsistent lineno tables *(Cromwell)*

(несогласованные номера строк таблицы)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

inconsistent storage class *(Синтаксический анализатор)*

(противоречивый класс памяти)

Объявления содержит конфликтные классы памяти. В объявлении может появиться только один класс памяти.

inconsistent symbol tables *(Cromwell)*

(несогласованные таблицы символов)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

inconsistent type *(Синтаксический анализатор)*

(противоречивый тип)

В объявлении может появиться только один базовый тип, поэтому сочетание таких комбинаций, как "int float", является недопустимым.

index offset too large *(Ассемблер)*

(смещение индекса слишком велико)

Смещение в индексированной адресной форме Z80 должно находится в диапазоне от -128 до 127.

initialisation syntax *(Синтаксический анализатор)*

(синтаксис инициализации)

Инициализация этого объекта синтаксически неверна. Проверьте правильность размещения и количество скобок и запятых.

initializer in 'extern' declaration *(Синтаксический анализатор)*

(инициализатор в объявлении 'extern')

Объявление, содержащее ключевое слово "extern", имеет инициализатор. Это переопределяет класс памяти "extern", так как для инициализации объекта необходимо определить его (т. е. выделить хранилище).

insufficient memory for macro def'n *(Ассемблер)*

(недостаточная память для определения макроса)

Не достаточно памяти для сохранения макроопределения.

integer constant expected *(Синтаксический анализатор)*

(ожидается целочисленная константа)

Двоеточие, появившееся после имени члена в объявлении структуры, указывает, что элемент является битовым полем. После двоеточия должна указываться целочисленная константа для определения количества бит в битовом поле.

integer expression required *(Синтаксический анализатор)*

(требуется целочисленное выражение)

В объявлении "enum" значения могут быть назначены членам, но выражение должно быть константой типа "int".

integral argument required *(Синтаксический анализатор)*

(требуется целочисленный аргумент)

Для этого спецификатора формата необходим целочисленный аргумент. Проверьте количество и порядок спецификаторов формата и соответствующие аргументы.

integral type required *(Синтаксический анализатор)*

(требуется целочисленный тип)

Этому оператору требуются операнды только целочисленного типа.

interrupt\_level should be 0 to 7 *(Синтаксический анализатор)*

(interrupt\_level должен быть от 0 до 7)

Директива pragma 'interrupt\_level' должна иметь аргумент от 0 до 7

invalid disable: \* *(Препроцессор)*

(недопустимое отключение: \*)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

invalid format specifier or type modifier *(Синтаксический анализатор)*

(недопустимый спецификатор формата или модификатор типа)

Описатель формата или модификатор в строке стиля printf() является недопустимым для данного формата.

invalid hex file: \*, line \* *(Cromwell)*

(недопустимый файл hex)

Указанный файл Hex содержит недопустимую строку.

invalid number syntax *(Ассемблер, Оптимизатор)*

(недопустимый синтаксис числа)

Недопустимый синтаксис числа. Это может быть, например, использование 8 или 9 в восьмеричном числе, или другие неправильные числа.

jump out of range *(Ассемблер)*

(переход вне диапазона)

Команде short jump ("JR") был задан адрес, который находится на расстоянии более 128 байтов от текущего местоположения. Вместо этого используйте опцию "JP".

label identifier expected *(Синтаксический анализатор)*

(ожидается идентификатор метки)

Идентификатор, обозначающий метку, должен появиться после "goto".

lexical error *(Ассемблер, Оптимизатор)*

(лексическая ошибка)

На входе был замечен нераспознанный символ или лексема.

library \* is badly ordered *(Компоновщик)*

(Библиотека \* плохо упорядочена)

Эта библиотека плохо упорядочена. Он по-прежнему будет правильно связываться, но она будет связываться быстрее, если улучшить порядок.

library file names should have .lib extension: \* *(Библиотекарь)*

(имена файлов библиотек должны иметь расширение .lib: \*)

Используйте расширение .lib при указании библиотеки.

line does not have a newline on the end *(Синтаксический анализатор)*

(Строка не имеет новой строки на конце)

В конце последней строки файла отсутствует новая строка (перевод строки, шестнадцатеричный OA). Некоторые редакторы будут создавать такие файлы, что может вызвать проблемы для включаемых файлов. Стандарт ANSI C требует, чтобы все исходные файлы состояли только из полных строк.

line too long *(Оптимизатор)*

(слишком длинная строка)

Эта строка слишком длинная. Он не будет вписываться в внутренние буферы компилятора. Для этого потребуется строка длиной более 1000 символов, поэтому обычно это происходит только в результате расширения макросов.

local illegal outside macros *(Ассемблер)*

(local не допустима вне макроса)

Директива "LOCAL" является законной только внутри макросов. В нем определяются локальные метки, которые будут уникальными для каждого вызова макроса.

local psect '\*' conflicts with global psect of same name *(Компоновщик)*

(Локальный psect \* конфликтует с глобальным psect с тем же именем)

Локальный psect не может иметь то же имя, что и глобальный psect.

logical type required *(Синтаксический анализатор)*

(требуется логический тип)

Выражение, используемое в качестве операнда для операторов "if", "while" или логических операторов типа ! и && должно быть скалярным целочисленным типом.

long argument required *(Синтаксический анализатор)*

(требуется аргумент типа long)

Для этого спецификатора формата требуется аргумент типа long. Проверьте количество и порядок спецификаторов формата и соответствующие аргументы.

macro \* wasn't defined *(Препроцессор)*

(макрос \* не определен)

Имя макроса, указанное в параметре -U препроцессора, изначально не определено и поэтому не может быть неопределенным.

macro argument after \* must be absolute *(Ассемблер)*

(аргумент макроса после \* должен быть абсолютным)

Аргумент после \* в макро-вызове должен быть абсолютным, поскольку он должен быть вычислен во время вызова макроса.

macro argument may not appear after local *(Ассемблер)*

(Аргумент макроса может не отображаться после local)

Список меток после директивы "LOCAL" не может содержать формальные параметры макроса.

macro expansions nested too deep *(Ассемблер)*

(макрорасширения вложены слишком глубоко)

Макрорасширения в ассемблере вложены слишком глубоко. Предел составляет 30 макросов и включаемых файлов, вложенных за один раз.

macro work area overflow *(Препроцессор)*

(переполнение рабочей области макросов)

Общая длина расширения макросов превысила размер внутренней таблицы. Эта таблица обычно имеет длину 8192 байта. Таким образом, любое расширение макросов не должно расширяться в общей сложности более 8 Кбайт.

member \* redefined *(Синтаксический анализатор)*

(член \* переопределен)

Это имя этого члена структуры или объединения уже используется в этой структуре или объединении.

members cannot be functions *(Синтаксический анализатор)*

(член не может быть функцией)

Член структуры или объединения не может быть функцией. Он может быть указателем на функцию. Правильный синтаксис для указателя функции требует использования круглых скобок для привязки звездочки ("\*") к имени указателя, например, "int (\*name)();".

metaregister \* can't be used directly *(Генератор кода)*

(метарегистр \* не может использоваться непосредственно)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

mismatched comparision *(Генератор кода)*

(несогласованное сравнение)

Сравнение делается между переменной или выражением и постоянным значением, которое не находится в диапазоне возможных значений для этого выражения, например, если вы сравните символ без знака с постоянным значением 300, результатом всегда будет ложь (не равно), так как символ без знака никогда не может равняться 300. Как 8 разрядное значение он может представлять только 0-255.

misplaced '?' or':', previous operator is \* *(Препроцессор)*

(неуместный '?' или ':', предыдущий оператор \*)

В выражении #if обнаружен оператор двоеточия, который не совпадает с соответствующим оператором ?. Проверьте круглые скобки и т.д.

misplaced constant in #if *(Препроцессор)*

(неуместная константа в #if)

Константа в выражении #if должна присутствовать только в синтаксически правильных местах. Эта ошибка, скорее всего, вызвана отсутствием оператора.

missing')' *(Синтаксический анализатор)*

(отсутствует ')' )

В этом выражении отсутствовала закрывающая скобка.

missing '=' in class spec *(Компоновщик)*

(отсутствует = в спецификации класса)

Спецификация класса требует знака =, например, -Ctext=ROM

missing']' *(Синтаксический анализатор)*

(отсутствует ']')

В этом выражении отсутствует закрывающая квадратная скобка.

missing arg to -a *(Синтаксический анализатор)*

(отсутствует аргумент -a)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

missing arg to -e *(Компоновщик)*

(отсутствует аргумент -e)

Имя файла ошибки должно быть указано после параметра -e компоновщика.

missing arg to -i *(Синтаксический анализатор)*

(отсутствует аргумент -i)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

missing arg to -j *(Компоновщик)*

(отсутствует аргумент -j)

Максимальное количество ошибок перед завершением должно быть определено после опции -j компоновщика.

missing arg to -q *(Компоновщик)*

(отсутствует аргумент -q)

Опция -Q компоновщика в качестве параметра требует тип машины.

missing arg to -u *(Компоновщик)*

(отсутствует аргумент -u)

Опции -U (не определен) требуется параметр, например, -U\_symbol

missing arg to -w *(Компоновщик)*

(отсутствует аргумент -w)

Опции -W (ширина листинга) нужен числовой параметр.

missing argument to 'pragma psect' *(Синтаксический анализатор)*

(отсутствует аргумент в 'pragma psect')

pragma 'psect' требует аргумент формы oldname=newname, где oldname - это существующее имя psect, известное компилятору, а newname - это требуемое новое имя. Пример: #pragma psect bss=battery

missing argument to 'pragma switch' *(Синтаксический анализатор)*

(отсутствует аргумент в 'pragma switch')

pragma 'switch' требует параметр auto, direct или simple.

missing basic type: int assumed *(Синтаксический анализатор)*

(отсутствует базовый тип: предполагается int)

Это объявление не включает базовый тип, поэтому предполагалось int. Это объявление не является неверным, однако предпочтительнее включить базовый тип, чтобы четко указать, что предполагается.

missing key in avmap file *(Компоновщик)*

(отсутствует ключ в файле avmap)

Поврежден файл, необходимый для создания файлов символов формата Avocet. Попробуйте переустановить компилятор.

missing memory key in avmap file *(Компоновщик)*

(отсутствует ключ памяти в файле avmap)

Поврежден файл, необходимый для создания файлов символов формата Avocet. Попробуйте переустановить компилятор.

missing name after pragma 'inline' *(Синтаксический анализатор)*

(отсутствует имя после pragma 'inline')

pragma 'inline' имеет синтаксис:

#pragma inline func\_name

где func\_name - имя функции, которая должна быть расширена до встроенного кода. Эта прагма влияет, только на функции, специально распознанные генератором кода.

missing name after pragma 'printf\_check' *(Синтаксический анализатор)*

(отсутствует имя после pragma 'printf\_check')

Директиве pragma 'printf\_check', которая включает проверку строки формата в стиле printf для функции, требуется имя функции, например

#pragma printf\_check sprintf

missing newline *(Препроцессор)*

(отсутствует новая строка)

В конце строки отсутствует символ новой строка. Каждая строка, включая последнюю строку, должна содержать в конце символ новой строки. Эта проблема обычно создается редакторами.

missing number after % in -p option *(Компоновщик)*

(отсутствует число после % в опции -p)

После оператора % в опции -p (для определения границ) должно быть число.

missing number after pragma 'pack' *(Синтаксический анализатор)*

(отсутствует число после pragma 'pack')

Директиве pragma 'pack' требуется десятичное число в качестве аргумента. Например

#pragma pack(1)

будет предотвращать выравнивание элементов структуры компилятором на что-либо, кроме границ одного байта. Используйте это с осторожностью, поскольку некоторые процессоры применяют выравнивание и будут работать неправильно, если выборка слов производится на нечетных границах (например, 68000, 8096).

missing number after pragma interrupt level *(Синтаксический анализатор)*

(отсутствует число после pragma interrupt\_level)

Директива pragma 'interrupt\_level' требует аргумент от 0 до 7.

missing processor name after -p *(Cromwell)*

(отсутствует имя процессора после –p)

Опция -p cromwell должна определить процессор.

mod by zero in #if, zero result assumed *(Препроцессор)*

(mod на ноль в #if, предполагается нулевой результат)

Операция "взятия остатка" в выражении #if имеет делитель нуля. Результат был принят равным нулю.

module \* defines no symbols *(Библиотекрь)*

(модуль \* не определяет символы)

В объектном файле модуля символы не найдены.

module has code below file base of \* *(Компоновщик)*

(в модуле имеется код, расположенный ниже базы файла \*)

Этот модуль имеет код ниже указанного адреса, но параметр -C использовался для указания того, что должен быть создан двоичный выходной файл, который сопоставляется с этим адресом. Это означало бы, что код из этого модуля должен быть помещен перед началом файла! Проверьте отсутствие директив psect в файлах ассемблера.

multi-byte constant \* isn't portable *(Препроцессор)*

(многобайтовая константа \* не является переносимой)

Многобайтовые константы не являются переносимыми и фактически будут отвергнуты более поздними проходами компилятора.

multiple free: \* *(Генератор кода)*

(несколько free: \*)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

multiply defined symbol \* *(Ассемблер, Компоновщик)*

(многократно определенный символ \*)

Этот символ был определен более чем в одном месте в этом модуле.

near function should be static *(Генератор кода)*

(near function должна быть static)

Близкая функция в модели с переключением банков должна быть статической, так как ее нельзя вызывать из другого модуля.

nested #asm directive *(Препроцессор)*

(вложенная директива #asm)

Вложенная директива #asm недопустима. Проверьте на отсутствие, или написание с ошибками директивы #endasm.

nested comments *(Препроцессор)*

(вложенные комментарии)

Это предупреждение выдается при обнаружении вложенных комментариев. Вложенный комментарий может указывать на то, что предыдущий маркер комментариев закрытия отсутствует или искажен.

no #asm before #endasm *(Препроцессор)*

(отсутствует #asm перед #endasm)

Обнаружен оператор #endasm, но предыдущий #asm не был найден.

no arg to -o *(Ассемблер)*

(нет аргумента для -o)

Ассемблеру требуется, чтобы после параметра "-O" был предоставлен аргумент имени выходного файла. Между параметром -O и именем файла не следует оставлять пробел.

no case labels *(Генератор кода)*

(отсутствуют метки case)

В операторе switch отсутствуют метки case.

no end record *(Компоновщик)*

(нет конца записи)

Этот объектный файл не имеет конечной записи. Это, вероятно, означает, что это не объектный файл.

no end record found *(Компоновщик)*

(конец записи не найден)

Объектный файл не содержал конечной записи. Вероятно, это означает, что файл поврежден или не является объектным файлом.

no file arguments *(Ассемблер)*

(отсутствуют файловые аргументы)

Ассемблер был вызван без параметров файла. Он ничего не может ассемблировать.

no identifier in declaration *(Синтаксический анализатор)*

(в описании отсутствует идентификатор)

В этом объявлении отсутствует идентификатор. Также эта ошибка может возникать там, где компилятор запутался в таких вещах, как отсутствие закрывающих скобок.

no input files specified *(Cromwell)*

(не указан входной файл)

Утилита Cromwell должна иметь входной файл для преобразования.

no memory for string buffer *(Синтаксический анализатор)*

()

Приложение P1 не удалось выделить память для самой длинной обнаруженной строки, поскольку оно пытается сортировать и объединить строки. Попытайтесь сократить количество или длину строк в этом модуле.

no output file format specified *(Cromwell)*

(не указан формат файла)

Утилите Cromwell должен быть определен выходной формат.

no psect specified for function variable/argument allocation *(Компоновщик)*

(не указана psect для выделения переменной/аргумента функции)

Это, вероятно, вызвано отсутствием корректного модуля времени выполнения startoff. Проверьте параметры компоновщика, или "Object Files..." (Объектные файлы...) в HPD.

no room for arguments (Preprocessor, Parser, Code Generator, Linker, Objtohex)

(нет памяти для аргументов)

Генератор кода не мог больше выделять память. Попытайтесь увеличить размер доступной памяти.

no space for macro def'n *(Ассемблер)*

(нет места для определения макросов)

Ассемблер исчерпал память.

no start record: entry point defaults to zero *(Компоновщик)*

(Отсутствует начальная запись: точка входа по умолчанию равна нулю)

Ни один из объектных файлов, переданных компоновщику, не содержал начальную запись. Начальный адрес программы установлен равным нулю. Это может быть безвредным, но рекомендуется определить начальный адрес в модуле запуска (startup) с помощью директивы "END".

no. of arguments redeclared *(Синтаксический анализатор)*

(количество аргументов переопределено)

Количество аргументов в объявлении этой функции не согласовано с предыдущим объявлением той же функции.

nodecount = \* *(Генератор кода)*

(счетчик узлов = \*)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

non-constant case label *(Генератор кода)*

(метка case не является константой)

Метка case в операторе switch имеет значение, которое не является константой.

non-prototyped function declaration: \* *(Синтаксический анализатор)*

(функция объявлена без прототипов: \*)

Функция была объявлена с использованием аргументов старого стиля (K&R). Предпочтительнее использовать объявления прототипов для всех функций. Если у функции нет аргументов, объявите ее, например, "int func(void)".

non-scalar types can't be converted *(Синтаксический анализатор)*

(нескалярный тип не может быть конвертирован)

Вы не можете преобразовать структуру, объединение или массив ни к чему больше. Вы можете преобразовать указатель на один из этих объектов, поэтому возможно, вы не учли амперсанд ("&").

non-void function returns no value *(Синтаксический анализатор)*

(не void функция не возвращает значение)

Функция, объявленная как возвращающая значение, имеет оператор return, который не указывает возвращаемое значение.

not a member of the struct/union \* *(Синтаксический анализатор)*

(не член структуры/объединения)

Этот идентификатор не является членом структуры или объединения, с которой он здесь используется.

not a variable identifier: \* *(Синтаксический анализатор)*

(идентификатор не является переменной)

Этот идентификатор не переменная. Это может быть некоторый объект другого вида, например, метка.

not an argument: \* *(Синтаксический анализатор)*

(отсутствует в аргументах)

Этот идентификатор, появившийся в объявлении атрибута аргумента типа K&R, не указан в круглых скобках после имени функции. Проверьте орфографию.

null format name *(Cromwell)*

(пустое имя формата)

Опции -I или -O утилиты Cromwell должны указывать формат файла.

object code version is greater than \* *(Компоновщик)*

(версия объектного кода больше, чем \*)

Версия объектного кода объектного модуля выше, чем самая высокая версия, с которой работает компоновщик. Убедитесь, что используется правильный компоновщик.

object file is not absolute *(Objtohex)*

(объектный файл не абсолютный)

В объектном файле, переданном objtohex, содержатся элементы перемещения. Это может означать, что это неправильный объектный файл, или что компоновщику или утилите objtohex были заданы недопустимые опции.

only functions may be qualified interrupt *(Синтаксический анализатор)*

(только функция может иметь квалификатор interrupt)

Квалификатор "interrupt" не может быть применен ни к чему кроме функции.

only functions may be void *(Синтаксический анализатор)*

(только функция может быть void)

Переменная не может быть "void". Только функция может быть "void".

only lvalues may be assigned to or modified *(Синтаксический анализатор)*

(только lvalue может быть присвоено или изменено)

Только lvalue (т.е. идентификатор или выражение, непосредственно обозначающее адресуемое хранилище) может быть присвоено или иным образом изменено. Приведенный тип не дает lvalue. Чтобы сохранить значение другого типа в переменную, возьмите адрес переменной, преобразуйте ее в указатель на нужный тип, затем разыменуйте этот указатель, например, "\*(int \*)&x = 1" является допустимым, тогда как "(int)x = 1" не является.

only modifier l valid with this format *(Синтаксический анализатор)*

(с этим форматом допустим только модификатор l)

Единственный модификатор, который допустим с этим форматом является l (для long).

only modifiers h and l valid with this format *(Синтаксический анализатор)*

(с этим форматом допустимы только модификаторы h и l)

Только модификаторы h (short) и l (long) допустимы с этим спецификатором формата printf().

only register storage class allowed *(Синтаксический анализатор)*

(разрешен только класс хранения register)

Единственным классом хранения, допустимым для параметра функции, является "register".

oops! -ve number of nops required! *(Ассемблер)*

(ой! -ve число требуется nops!)

Произошла внутренняя ошибка. Свяжитесь с HI-TECH.

operand error *(Ассемблер, Оптимизатор)*

(ошибка операнда)

Операнд для этого кода операции недопустим. Проверьте руководство по ассемблеру для уточнения правильной формы операндов этой инструкции.

operands of \* not same pointer type *(Синтаксический анализатор)*

(операнды \* имеют разные типы указателей)

Операнды этого оператора имеют указатели разных типов. Вероятно, это означает, что вы использовали неправильный указатель, но если код действительно то, что вы намеревались, используйте приведение типа для подавления сообщения об ошибке.

operands of \* not same type *(Синтаксический анализатор)*

(операнды \* имеют разные типы)

Операнды этого оператора имеют разные указатели. Вероятно, это означает, что вы использовали неправильную переменную, но если код на самом деле то, что вы намеревались, используйте приведение типа для подавления сообщения об ошибке.

operator \* in incorrect context *(Препроцессор)*

(оператор \* в неправильном контексте)

Обнаружен не правильно размещенный оператор в выражении #if, например, два бинарных оператора не разделены значением.

out of far memory *(Генератор кода)*

(исчерпана дальняя память)

Компилятор исчерпал дальнюю память. Попробуйте удалить резидентные программы (TSR) и т.д. Если система поддерживает отображаемую память EMS, компилятор сможет использовать до 64 КБ, поэтому если она не включена, попробуйте включить EMS.

out of memory *(Генератор кода, Ассемблер, Оптимизатор)*

(нехватка памяти)

У компилятора закончилась память. Если у вас загружены ненужные TSR, удалите их. Если вы используете компилятор из другой программы, попробуйте запустить его непосредственно из командной строки. Аналогично, если вы используете HPD, попробуйте вместо этого использовать драйвер компилятора командной строки.

out of memory allocating \* blocks of \* *(Компоновщик)*

(недостаточно распределяемой памяти \* блоков \*)

Память была необходима для расширения массива, но была недоступна.

out of memory for assembler lines *(Оптимизатор)*

(недостаточно памяти для ассемблерных строк)

Оптимизатор исчерпал память для хранения ассемблерных строк, например, строки #asm из источника C или исходные строки комментариев C. Уменьшите размер функции.

out of near memory *(Генератор кода)*

(исчерпана ближняя память)

Компилятор не хватает памяти. Вероятно, это связано с слишком большим количеством имен символов. Попробуйте разделить программу или уменьшите количество неиспользуемых символов в файлах заголовков и т.д.

out of space in macro \* arg expansion *(Препроцессор)*

(исчерпана память в макро \* при расширении аргументов)

Аргумент макроса превысил длину внутреннего буфера. Этот буфер обычно имеет длину 4096 байт.

output file cannot be also an input file *(Компоновщик)*

(выходной файл не может быть также входным файлом)

Компилятор обнаружил попытку записать свой выходной файл в один из своих входных файлов. Это не может быть сделано, потому что ему необходимо одновременно читать и записывать входные и выходные файлы.

page width must be >= \* *(Ассемблер)*

(ширина страницы должна быть >= \*)

Ширина страницы листинга должна быть не менее \* символов. Любое меньшее число не позволит создать правильно отформатированный листинг.

phase error *(Ассемблер)*

(ошибка фазы)

Ассемблер вычислил различное значение символа на двух различных проходах. Это, вероятно, связано с причудливым использованием макросов или условной сборки.

phase error in macro args *(Ассемблер)*

(ошибка фазы в макро аргументах)

Ассемблер обнаружил различие в определении символа на первом и последующем проходе.

phase error on temporary label *(Ассемблер)*

(ошибка фазы во временной метке)

Ассемблер обнаружил разницу в определении символа на первом и последующем проходе.

pointer required *(Синтаксический анализатор)*

(требуется указатель)

Здесь требуется указатель. Это часто означает, что вы использовали "->" со структурой, а не указатель на структуру.

pointer to \* argument required *(Синтаксический анализатор)*

(требуется указатель на аргумент \*)

Требуется указатель аргумента для этого спецификатора формата. Проверьте количество и порядок спецификаторов формата и соответствующие аргументы.

pointer to non-static object returned *(Синтаксический анализатор)*

(требуется указатель на не статический объект)

Эта функция возвращает указатель на не статическую (например, автоматическую) переменную. Вероятно, это ошибка, так как память, связанная с автоматическими переменными, становится недействительной, после возврата из функции.

portion of expression has no effect *(Генератор кода)*

(часть выражения не имеет эффекта)

Часть этого выражения не имеет побочных эффектов и не влияет на значение выражения.

possible pointer truncation *(Синтаксический анализатор)*

(возможно усечение указателя)

Указатель, квалифицированный как "far" (дальний), назначен указателю по умолчанию или указателю, квалифицированному как "near" (близкий), или указателю по умолчанию, квалифицированному как "near". Это может привести к усечению указателя и потере информации в зависимости от используемой модели памяти.

preprocessor assertion failure *(Препроцессор)*

(сбой утверждения препроцессора)

Аргумент директивы #assert препроцессора оценивается равным нулю. Это ошибка, вызванная программистом.

probable missing '}' in previous block *(Синтаксический анализатор)*

(вероятное отсутствие '}' в предыдущем блоке)

Компилятор столкнулся с тем, что выглядит как функция или другое объявление, но предыдущая функция не была завершена закрывающей фигурной скобкой. Это, вероятно, означает, что закрывающая фигурная скобка была опущена где-то в предыдущей функции, хотя она вполне может быть не последней.

psect \* cannot be in classes \* *(Компоновщик)*

(psect \* не может быть в классе)

psect не может быть более чем в одном классе. Это либо из-за модулей ассемблера с конфликтующими опциями class=, либо использования опции -C компоновщика.

psect \* memory delta redefined: \*/\* *(Компоновщик)*

(psect \* переопределена дельта памяти: \*/\*)

Глобальная psect была определена с двумя разными дельтами.

psect \* memory space redefined: \*/\* *(Компоновщик)*

(psect \* переопределено пространство памяти: \*/\*)

Глобальная psect была определена в двух разных пространствах памяти. Либо переименуйте один из этих psect, либо, если они будут тем же самым psect, поместите их в одно и то же пространство памяти, используя флаг SPACE psect.

psect \* not loaded on \* boundary *(Компоновщик)*

(psect \* не загружается на \* границу)

Этот psect имеет требование переместимости, которое не удовлетворяет адресу загрузки, указанному в опции -P. Например, если psect должен находиться на границе 4K байтов, вы не сможете запустить его на 100H.

psect \* not relocated on \* boundary *(Компоновщик)*

(psect \* не перемещается на \* границу)

Этот psect не перемещается на требуемую границу. Проверьте перемещаемость psect и при необходимости исправьте параметр -p.

psect \* not specified in -p option *(Компоновщик)*

(psect \* не указан в опции -p)

Эта psect не была определена в опции -P или -A компоновщика. Она была скомпонована в конце программы, что, вероятно, не там где нужно.

psect \* re-orged *(Компоновщик)*

(psect \* переупорядочена)

Этой psect определили начальный адрес несколько раз.

psect \* selector value redefined *(Компоновщик)*

(значение селектора psect \* переопределено)

Значение селектора для этой psect было определено несколько раз.

psect \* type redefined: \* *(Компоновщик)*

(тип psect \* переопределен: \*)

Эта psect имела свой тип, определенный по-разному разными модулями. Это, вероятно, означает, что вы пытаетесь связать несовместимые объектные модули, например, связываете код 386 плоской модели с кодом 8086 в реальном режиме.

psect exceeds address limit: \* *(Компоновщик)*

(psect превышает предельный адрес: \*)

Максимальный адрес psect превышает границу, установленную для нее, используя флаг LIMIT psect.

psect exceeds max size: \* *(Компоновщик)*

(psect превышает макс. размер: \*)

В psect содержится больше байтов, чем максимально допустимое значение, указанное с помощью флага SIZE psect.

psect is absolute: \* *(Компоновщик)*

(psect абсолютный: \*)

Эта psect абсолютная и не должна определять адрес в опции -P.

psect may not be local and global *(Ассемблер)*

(psect не может быть локальным и глобальным)

psect не может быть объявлена локальной, если она уже объявлена глобальной (по умолчанию).

psect origin multiply defined: \* *(Компоновщик)*

(начало psect определено несколько раз: \*)

Начало этой psect определено более одного раза.

psect property redefined *(Ассемблер)*

(Свойства psect переопределены)

Свойства psect были определены в нескольких местах по разному.

psect reloc redefined *(Ассемблер)*

(Перемещение psect переопределено)

Перемещение этой psect было определено по-разному в двух или более местах.

psect selector redefined *(Компоновщик)*

(селектор psect переопределен)

Селектор, связанный с этим psect, был определен по-разному в двух или более местах.

psect size redefined *(Ассемблер)*

(размер psect переопределен)

Максимальный размер этого psect был определен по-разному в двух или более местах.

qualifiers redeclared *(Синтаксический анализатор)*

(квалификатор переопределен)

Эта функция имеет различные квалификаторы в различных объявлениях.

read error on \* *(Компоновщик)*

(ошибка чтения в \*)

Компоновщик обнаружил ошибку, пытаясь прочитать этот файл.

record too long *(Objtohex)*

(запись слишком длинная)

Это указывает на то, что объектный файл не является допустимым файлом HI-TECH.

record too long: \* *(Компоновщик)*

(запись слишком длинная: \*)

Объектный файл содержит запись с недопустимым размером. Вероятно, это означает, что файл поврежден или не является объектным файлом.

recursive function calls: *(Компоновщик)*

(рекурсивный вызов функции:)

Эти функции (или функция) вызывают друг друга рекурсивно. Одна или несколько из этих функций имеют статически распределенные локальные переменные (скомпилированный стек). Чтобы избежать рекурсии, используйте зарезервированное слово reentrant (если оно поддерживается этим компилятором) или измените код, чтобы избежать рекурсии.

recursive macro definition of \* *(Препроцессор)*

(рекурсивное определение макроса \*)

Названный макрос был определен таким образом, что его расширение вызывает рекурсивное расширение себя!

redefining macro \* *(Препроцессор)*

(переопределение макроса \*)

Указанный макрос переопределяется, чем-то иным, чем исходное определение. Если вы хотите намеренно переопределить макрос, сначала используйте #undef, чтобы удалить исходное определение.

redundant & applied to array *(Синтаксический анализатор)*

(избыточный & примененный к массиву)

Оператор адреса "&" применяется к массиву. Так как использование имени массива дает его адрес в любом случае, это ненужно и было проигнорировано.

refc == 0 *(Ассемблер, Оптимизатор)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

regused - bad arg to g *(Генератор кода)*

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

relocation error *(Ассемблер, Оптимизатор)*

(ошибка перемещения)

Невозможно сложить две перемещаемые величины. Константа может быть добавлена к перемещаемому значению, и два перемещаемых адреса в одной и той же psect могут быть вычтены. Абсолютное значение должно использоваться в различных местах, где ассемблер должен знать значение во время ассемблирования.

relocation offset \* out of range \* *(Компоновщик)*

(смещение перемещения \* вне диапазона \*)

Объектный файл содержит запись перемещения с смещением перемещения вне диапазона предыдущей текстовой записи. Это означает, что объектный файл, вероятно, поврежден.

relocation too complex *(Ассемблер)*

(перемещение слишком сложное)

Сложное перемещение в этом выражении слишком велико для вставки в объектный файл.

remsym error *(Ассемблер)*

(ошибка remsym)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

replace what ? *(Библиотекарь)*

(заменить что ?)

Библиотекарь требует перечисления одного или несколько модулей для замены при использовании ключа 'r'.

rept argument must be >= 0 *(Ассемблер)*

(аргумент rept должен быть >=0)

Аргумент директивы "REFT" должен быть больше нуля.

seek error: \* *(Компоновщик)*

(ошибка поиска)

При записи выходного файла компоновщик не смог выполнить поиск.

segment \* overlaps segment \* *(Компоновщик)*

(сегмент \* перекрывает сегмент \*)

У названных сегментов есть перекрывающийся код или данные. Проверьте адреса, присваиваемые опцией "-P".

signatures do not match: \* *(Компоновщик)*

(сигнатуры не совпадают: \*)

В различных модулях указанная функция имеет разные сигнатуры. Это означает, что она была объявлена иначе, например, она может иметь прототип в одном модуле, а не другом. Проверьте, какие объявления функции видны в двух указанных модулях и убедитесь, что они совместимы.

signed bitfields not supported *(Синтаксический анализатор)*

(битовые поля со знаком, не поддерживаются)

Поддерживаются только битовые поля без знака. Если объявленное битовое поле имеет тип "int", компилятор по-прежнему рассматривает его как беззнаковое.

simple integer expression required *(Синтаксический анализатор)*

(требуется простое целочисленное выражение)

После оператора "@", используемого для связывания абсолютного адреса с переменной требуется простое целочисленное выражение.

simple type required for \* *(Синтаксический анализатор)*

(требуется простой тип для \*)

В качестве операнда этому оператору требуется простой тип (т.е. не массив или структура).

sizeof external array \* is zero *(Синтаксический анализатор)*

(sizeof внешнего массива \* равен нулю)

sizeof внешнего массива равен нулю. Вероятно, это связано с тем, что массив не имеет явного измерения в объявлении extern.

sizeof yields 0 *(Генератор кода)*

(sizeof возвращает 0)

Генератор кода взял размер объекта и нашел его равным нулю. Это почти наверняка указывает на ошибку в объявлении указателя, например, вы можете объявить указатель на массив нулевой длины. В целом указатели на массивы мало используются. Если требуется указатель на массив объектов неизвестной длины, требуется только указатель на один объект, который затем может быть проиндексирован или инкрементирован.

static object has zero size: \* *(Генератор кода)*

(статический объект имеет нулевой размер: \*)

Статический объект объявлен, но имеет нулевой размер.

storage class illegal *(Синтаксический анализатор)*

(неверный класс хранения)

Члену структуры или объединению нельзя задать класс хранения. Его класс хранения определяется классом хранения структуры.

storage class redeclared *(Синтаксический анализатор)*

(класс хранения переопределен)

Переменная или функция была повторно объявлена с другим классом хранения. Это может происходить в тех случаях, когда существуют два противоречащих друг другу объявления или когда за неявным объявлением следует фактическое объявление.

strange character \* after ## *(Препроцессор)*

(странный символ \* после ##)

После оператора объединения токенов ##, был обнаружен символ, который не является ни буквой, ни цифрой. Поскольку результатом этого оператора должен быть легальный токен, операнды должны быть токенами, содержащими только буквы и цифры.

strange character after # \* *(Препроцессор)*

(странный символ после # \*)

Существует неожиданный символ после #.

string concatenation across lines *(Синтаксический анализатор)*

(Объединение строк через строки)

Строки на двух строках будут объединены. Убедитесь, что это желаемый результат.

string expected *(Синтаксический анализатор)*

(ожидается строка)

Операнд в операторе "asm" должен быть строкой, заключенной в круглые скобки.

string lookup failed in coff:get\_string() *(Cromwell)*

(Сбой поиска строки в coff: get\_string())

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

string too long *(Ассемблер)*

(слишком длинная строка)

Эта строка слишком длинная. Сократите ее.

struct/union member expected *(Синтаксический анализатор)*

(ожидается член структуры/объединения)

За точкой (".") или стрелкой ("->") должно следовать имя члена структуры или объединения.

struct/union redefined: \* *(Синтаксический анализатор)*

(структура/объединение перереопределено: \*)

Структура или объединение были определены несколько раз.

struct/union required *(Синтаксический анализатор)*

(требуется структура/объединение )

Перед точкой (".") требуется идентификатор структуры или объединения.

struct/union tag or '{' expected *(Синтаксический анализатор)*

(ожидается тег структуры объединения или '{')

Идентификатор, обозначающий структуру или объединение или открывающая фигурная скобка, должны следовать за ключевым словом "struct" или "union".

symbol \* cannot be global *(Компоновщик)*

(символ \* не может быть глобальным)

Ошибка в объектном файле, где локальный символ был объявлен глобальным. Это либо недопустимый объектный файл, либо внутренняя ошибка компоновщика. Попробуйте повторно создать объектный файл.

symbol \* has erroneous psect: \* *(Компоновщик)*

(символ \* имеет ошибочный psect: \*)

Ошибка в объектном файле, где символ имеет недопустимую psect. Это либо недопустимый объектный файл, либо внутренняя ошибка компоновщика. Попробуйте повторно создать объектный файл.

symbol \* not defined in #undef *(Препроцессор)*

(символ \* не определен в #undef)

Символ, указанный как аргумент #undef, еще не определен. Это предупреждение, но его можно избежать, включив #undef в блок #ifdef ... #endif.

syntax error *(Ассемблер, Оптимизатор)*

(синтаксическая ошибка)

Обнаружена синтаксическая ошибка. Это может быть вызвано рядом причин.

syntax error in -a spec *(Компоновщик)*

(синтаксическая ошибка в спецификации -a)

Недопустимая спецификация -A. Корректная спецификация -A должна быть примерно такой:

-AROM=1000h-lFFFh

syntax error in checksum list *(Компоновщик)*

(синтаксическая ошибка в списке контрольных сумм)

Синтаксическая ошибка в списке контрольных сумм, считанном компоновщиком. Список контрольных сумм считывается со стандартного ввода компоновщиком в ответ на параметр. Повторно прочитайте руководство по списку контрольных сумм.

syntax error in local argument *(Ассемблер)*

(синтаксическая ошибка в локальном аргументе)

В локальном аргументе имеется синтаксическая ошибка.

text does not start at 0 *(Компоновщик)*

(текст не начинается с 0)

Код в некоторых вещах должен начинаться с нуля. Здесь это не так.

text offset too low *(Компоновщик)*

(смещение текста слишком маленькое)

Смещение текста слишком маленькое

text record has bad length: \* *(Компоновщик)*

(текстовая запись имеет неверную длину: \*)

Ошибка в объектном файле. Это либо недопустимый объектный файл, либо внутренняя ошибка компоновщика. Попробуйте повторно создать объектный файл.

text record has length too small: \* *(Компоновщик)*

(текстовая запись имеет слишком малую длину: \*)

Это указывает на то, что объектный файл не является допустимым файлом HI-TECH.

this function too large - try reducing level of optimization *(Генератор кода)*

(эта функция слишком велика - попробуйте уменьшить уровень оптимизации)

При использовании переключателя -Og (глобальная оптимизация) обнаружена большая функция. Попробуйте выполнить повторную компиляцию без глобальной оптимизации или уменьшите размер функции.

this is a struct *(Синтаксический анализатор)*

(это - структура)

Этот идентификатор, следующий за ключевым словом "union" или "enum", уже является тегом для структуры и поэтому должен следовать только за ключевым словом "struct".

this is a union *(Синтаксический анализатор)*

(это - объединение)

Этот идентификатор, следующий за ключевым словом "struct" или "enum", уже является тегом для объединения и поэтому должен следовать только за ключевым словом "union".

this is an enum *(Синтаксический анализатор)*

(это - перечисление)

Этот идентификатор, следующий за ключевым словом "struct" или "union", уже является тегом для перечисляемого типа и поэтому должен следовать только за ключевым словом "enum".

too few arguments *(Синтаксический анализатор)*

(слишком мало аргументов)

Эта функция требует больше аргументов, чем в этом вызове.

too few arguments for format string *(Синтаксический анализатор)*

(слишком мало аргументов для строки формата)

Слишком мало аргументов для этой строки формата. Это приведет к тому, что значение мусора будет напечатано или преобразовано во время выполнения.

too many (\*) enumeration constants *(Синтаксический анализатор)*

(слишком много перечисляемых констант)

В перечислимом типе имеется слишком много констант перечисления. Максимальное число перечисляемых констант, допустимых в перечисляемом типе, составляет 512.

too many (\*) structure members *(Синтаксический анализатор)*

(слишком много членов структуры (\*))

Слишком много членов в структуре или объединении. Максимальное число членов, разрешенное в одной структуре или объединении, составляет 512.

too many address spaces - space \* ignored *(Компоновщик)*

(слишком много адресных пространств – пространство \* игнорировано)

Предельное число адресных пространств в настоящее время составляет 16.

too many arguments *(Синтаксический анализатор)*

(слишком много аргументов)

Эта функция не принимает столько аргументов, сколько существует здесь.

too many arguments for format string *(Синтаксический анализатор)*

(слишком много аргументов в строке формата)

Для этой строки формата слишком много аргументов. Это безопасно, но может представлять собой неправильную строку формата.

too many arguments for macro *(Препроцессор)*

(слишком много аргументов в макросе)

Макрос может иметь только 31 параметр в соответствии со стандартом C.

too many arguments in macro expansion *(Препроцессор)*

(слишком много аргументов в макро расширении)

При вызове макроса было задано слишком много аргументов. Максимально допустимое число составляет 31.

too many cases in switch *(Генератор кода)*

(в switch слишком много case)

В этом операторе switch слишком много меток case. Максимальное допустимое число меток case в любой операторе switch составляет 511.

too many comment lines – discarding *(Ассемблер)*

(слишком много строк комментариев -- отбрасывание)

Компилятор генерирует ассемблерный код со встроенными комментариями, но эта функция настолько велика, что создается избыточное число комментариев исходной строки. Они подавляются таким образом, что оптимизатор не будет загружать строки комментариев.

too many errors *(Все программы)*

(слишком много ошибок)

Было много ошибок, которые выдал компилятор. Исправьте первые ошибки, и многие из них, вероятно, исчезнут.

too many file arguments, usage: cpp [input [output]] *(Препроцессор)*

(слишком много файлов в качестве аргументов, используйте cpp [input [output])

CPP следует вызывать не более чем с двумя аргументами файлов.

too many files in coff file *(Cromwell)*

(слишком много файлов в файле coff)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

too many include directories *(Препроцессор)*

(слишком много включаемых каталогов)

Для препроцессора можно указать не более 7 каталогов для поиска включаемых файлов.

too many initializers *(Синтаксический анализатор)*

(слишком много инициализаторов)

Слишком много инициализаторов для этого объекта. Сопоставьте количество инициализаторов с определением объекта (массив или структура).

too many input files *(Cromwell)*

(слишком много входных файлов)

Слишком много входных файлов было указано Cromwell для преобразования.

too many macro parameters *(Ассемблер)*

(слишком много параметров макроса)

В этом определении макроса слишком много макропараметров.

too many nested #\* statements *(Препроцессор)*)

(слишком много вложенных #\* операторов)

Блоки #if, #ifdef и т.д. могут быть вложены максимум 32 раза.

too many nested #if statements *(Препроцессор)*

(слишком много вложенных операторов #if)

Блоки #if, #ifdef и т.д. могут быть вложены максимум 32 раза.

too many output files *(Cromwell)*

(слишком много выходных файлов)

Слишком много выходных файлов форматов указано программе Cromwell.

too many psect class specifications *(Компоновщик)*

(слишком много спецификаций класса psect)

Слишком много спецификаций класса psect (опций -C)

too many psect pragmas *(Генератор кода)*

(слишком много pragma psect)

Было использовано слишком много директив "pragma psect".

too many psects *(Ассемблер)*

(слишком много psect)

Слишком много psect! Боже, какая программа!

too many qualifier names *(Синтаксический анализатор)*

(слишком много имен квалификаторов)

Указано слишком много имен квалификаторов.

too many relocation items *(Objtohex)*

(слишком много перемещаемых элементов)

Программа objtohex заполнила таблицу. Эта программа просто слишком сложная!

too many segment fixups *(Objtohex)*

(слишком много исправлений сегментов)

В объектном файле, предоставленном objtohex, слишком много исправлений сегментов.

too many segments *(Objtohex)*

(слишком много сегментов)

Слишком много сегментов в объектном файле, предоставленном objtohex.

too many symbols *(Ассемблер)*

(слишком много символов)

Для таблицы символов ассемблеров слишком много символов. Сократите количество символов в программе. Если эту ошибку выдает компоновщик, рекомендуется изменить некоторые глобальные символы на локальные.

too many symbols (\*) *(Компоновщик)*

(слишком много символов в (\*))

В таблице символов слишком много символов, которые имеют предел \* символов. Измените некоторые глобальные символы на локальные, чтобы уменьшить количество символов.

too many symbols in \* *(Оптимизатор)*

(слишком много символов в \*)

В указанной функции слишком много символов. Уменьшите размер функции.

too many temporary labels *(Ассемблер)*

(слишком много временных меток)

В этом файле ассемблера слишком много временных меток. Ассемблер допускает максимум 2000 временных меток.

too much indirection *(Синтаксический анализатор)*

(слишком много косвенности)

Объявление указателя может иметь только 16 уровней косвенности.

too much pushback *(Препроцессор)*

(слишком много pushback)

Эта ошибка не должна возникать и представляет собой внутреннюю ошибку в препроцессоре.

type conflict *(Синтаксический анализатор)*

(конфликт типов)

Операнды этого оператора имеют несовместимые типы.

type modifier already specified *(Синтаксический анализатор)*

(Модификатор типа уже указан)

Этот модификатор типа уже указан в этом типе.

type modifiers not valid with this format *(Синтаксический анализатор)*

(модификаторы типа недопустимы с этим форматом)

Модификаторы типа не могут использоваться с этим форматом.

type redeclared *(Синтаксический анализатор)*

(Тип повторно объявлен)

Тип этой функции или объекта был повторно объявлен. Это может произойти из-за двух несовместимых объявлений или из-за того, что за неявным объявлением следует несовместимое объявление.

type specifier reqd. for proto arg *(Синтаксический анализатор)*

(Спецификатор типа необходим для прототипированного аргумента)

Спецификатор типа необходим для прототипированного аргумента. Недопустимо иметь только идентификатор.

unable to open list file \* *(Компоновщик)*

(не удалось открыть файл списка \*)

Указанный файл списка не может быть открыт.

unbalanced paren's, op is \* *(Препроцессор)*

(неуравновешенные скобки \*)

Оценка выражения #if обнаружила несогласованные круглые скобки. Проверьте выражение на правильность расстановок скобок.

undefined \*: \* *(Синтаксический анализатор)*

(не определен \*: \*)

Это внутренняя ошибка компилятора. Для уточнения деталей свяжитесь с технической поддержкой HI-TECH Software.

undefined enum tag: \* *(Синтаксический анализатор)*

(не определен тег enum: \*)

Этот тег перечисления не определен.

undefined identifier: \* *(Синтаксический анализатор)*

(неопределенный идентификатор: \*)

Этот символ используется в программе, но не определен и не объявлен. Проверьте наличие орфографических ошибок.

undefined shift (\* bits) *(Генератор кода)*

(неопределенный сдвиг (\* бит))

Предпринята попытка сдвига значения на число битов, равное или больше, чем число битов в типе данных, например, сдвиг long на 32 бита. Это приведет к неопределенному результату на многих процессорах. Это непереносимый код и помечается как имеющий неопределенные результаты по стандарту C.

undefined struct/union *(Синтаксический анализатор)*

(неопределенная структура/объединение)

Этот тег структуры или объединения не определен. Проверьте орфографию и т.д.

undefined struct/union: \* *(Синтаксический анализатор)*

(неопределенная структура/объединение: \*)

Этот тег структуры или объединения не определен. Проверьте орфографию и т.д.

undefined symbol \* *(Ассемблер)*

(неопределенный символ \*)

Названный символ не определен и не был определен как "GLOBAL".

undefined symbol \* in #if, 0 used *(Препроцессор)*

(не определен символ \* в #if, используется 0)

Символ в выражении #if не был определен макросом препроцессора. Для целей этого выражения его значение принимается равным нулю.

undefined symbol in fnaddr record: \* *(Компоновщик)*

(не определен символ в записи fnaddr: \*)

Компоновщик нашел неопределенный символ в записи fnaddr для не реентерабельной функции.

undefined symbol in fnbreak record: \* *(Компоновщик)*

(не определен символ в записи fnbreak: \*)

Компоновщик нашел неопределенный символ в записи fnbreak для не реентерабельной функции.

undefined symbol in fncall record: \* *(Компоновщик)*

(не определен символ в записи fncall: \*)

Компоновщик нашел неопределенный символ в записи fncall для не реентерабельной функции.

undefined symbol in fnindir record: \* *(Компоновщик)*

(не определен символ в записи fnindir: \*)

Компоновщик нашел неопределенный символ в записи fnindir для не реентерабельной функции.

undefined symbol in fnroot record: \* *(Компоновщик)*

(не определен символ в записи fnroot: \*)

Компоновщик нашел неопределенный символ в записи fnroot для не реентерабельной функции.

undefined symbol in fnsize record: \* *(Компоновщик)*

(неопределенный символ в записи fnsize: \*)

Компоновщик нашел неопределенный символ в записи fnsize для не реентерабельной функции.

undefined symbol: *(Ассемблер, Компоновщик)*

(неопределенный символ:)

Следующий символ не определен во время компоновки. Это может быть вызвано орфографической ошибкой или невозможностью связывания соответствующего модуля.

undefined symbols: *(Компоновщик)*

(неопределенный символы:)

Далее следует список символов, которые не были определены во время компоновки.

undefined temporary label *(Ассемблер)*

(неопределенная временная метка)

Ссылка на временную метку, которая не определена. Обратите внимание, что временная метка должна иметь число >= 0.

undefined variable: \* *(Синтаксический анализатор)*

(неопределенная переменная: \*)

Эта переменная использовалась, но не определялась в этом месте.

unexpected end of file *(Компоновщик)*

(неожиданный конец файла)

Это, вероятно, означает, что  объектный файл был усечен из-за отсутствия дискового пространства.

unexpected eof *(Синтаксический анализатор)*

(неожиданный конец файла)

Неожиданно был обнаружен конец файла. Проверьте синтаксис.

unexpected text in #control line ignored *(Препроцессор)*

(неожиданный текст в строке #control проигнорирован)

Это предупреждение возникает, когда на конце строки управления появляются дополнительные символы, например

#endif something

Строка "something" будет проигнорирован, но выдано предупреждение. Предпочтительно (и в соответствии со стандартом С) включить "something" в качестве комментария, например

#endif /\* something \*/

unexpected \ in #if *(Препроцессор)*

(неожиданный \ в #if)

Обратная косая черта недопустима в инструкции #if.

unknown 'with' psect referenced by psect \* *(Компоновщик)*

(неизвестный psect 'with', на который ссылается psect \*)

Указанная psect была помещена с помощью psect с использованием флага 'with'. psect, в которую он был помещен, не существует.

unknown complex operator \* *(Компоновщик)*

(неизвестный сложный оператор \*)

Ошибка в объектном файле. Это либо недопустимый объектный файл, либо внутренняя ошибка компоновщика. Попробуйте повторно создать объектный файл.

unknown fnrec type \* *(Компоновщик)*

(неизвестный тип fnrec \*)

Это означает, что объектный файл не является допустимым объектным файлом HI-TECH.

unknown format name '\*' *(Cromwell)*

(неизвестный формат имени '\*')

Формат вывода, указанный для Cromwell, неизвестен.

unknown option \* *(Препроцессор)*

(неизвестная опция \*)

Эта опция препроцессора не распознана.

unknown pragma \* *(Синтаксический анализатор)*

(неизвестная pragma \*)

Была обнаружена неизвестная директива pragma.

unknown predicate\* *(Генератор кода)*

(неизвестный предикат \*)

Внутренняя ошибка – обращайтесь в HI-TECH.

unknown psect *(Оптимизатор)*

(неизвестная psect)

Файл ассемблера, прочитанный оптимизатором, содержит неизвестную psect.

unknown psect: \* *(Компоновщик, Оптимизатор)*

(неизвестный psect: \*)

Этот psect был указан в опции -P, но не определен ни в одном модуле в программе.

unknown qualifier '\*' given to –a *(Синтаксический анализатор)*

(неизвестный спецификатор '\*' заданный –a)

Обратитесь в службу технической поддержки HI-TECH Software за подробной информацией.

unknown qualifier '\*' given to –I *(Синтаксический анализатор)*

(неизвестный спецификатор '\*' заданный –I)

Это внутренняя ошибка компилятора. Обратитесь в службу технической поддержки HI-TECH Software за подробной информацией.

unknown record type: \* *(Компоновщик)*

(неизвестный тип записи: \*)

Компоновщиком был прочитан недопустимый объектный модуль. Он либо поврежден, либо не является объектным файлом.

unknown register name \* *(Компоновщик)*

(неизвестное имя регистра \*)

Это внутренняя ошибка компилятора. Обратитесь в службу технической поддержки HI-TECH Software за подробной информацией.

unknown symbol type \* *(Компоновщик)*

(неизвестный тип символа \*)

Обнаруженный тип символа неизвестен этому компоновщику. Убедитесь, что используется правильный компоновщик.

unreachable code *(Синтаксический анализатор)*

(недостижимый код)

Этот раздел кода никогда не будет выполняться, так как не существует пути выполнения, по которому его можно было бы достичь. Ищите отсутствующие операторы "break" в структуре элемента управления, например "while" или "for".

unreasonable matching depth *(Генератор кода)*

(неразумная глубина соответствия)

Это внутренняя ошибка компилятора. Обратитесь в службу технической поддержки HI-TECH Software за подробной информацией.

unrecognized option to -z: \* *(Генератор кода)*

(непризнанная опция -z: \*)

Генератору кода была передана опция -Z, которую он не понимает. Это не должно происходить, если он вызван стандартным драйвером.

unrecognized qualifer name after 'strings' *(Синтаксический анализатор)*

(нераспознанное имя квалификатора после 'strings')

Директива pragma 'strings' требует список допустимых имен квалификаторов. Например,

#pragma strings const code

добавили бы const и code к текущим квалификаторам строки. Если никакие квалификаторы не определены, вся квалификация будет удалена из последующих строк. Имена квалификаторов должны быть распознаны компилятором.

unterminated #if[n][def] block from line \* *(Препроцессор)*

(незавершенный блок #if[n][def] в строке \*)

Блок #if или подобный не был завершен соответствующим #endif. Номер строки указывает строку, на которой начался блок #if.

unterminated comment in included file *(Препроцессор)*

(незавершенный комментарий во включаемом файле)

Комментарии, начатые внутри включаемого файла, должны заканчиваться внутри включаемого файла.

unterminated macro arg *(Ассемблер)*

(незавершенный аргумент макро)

Аргумент для макроса не завершается. Обратите внимание, что для цитирования аргументов макрокоманд используются угловые скобки ("< >").

unterminated string *(Ассемблер, Оптимизатор)*

(незавершенная строка)

Строковая константа не содержит заключительной кавычки.

unterminated string in macro body *(Препроцессор, Ассемблер)*

(незавершенная строка в теле макроса)

Определение макроса содержит строку, которой не хватает заключительной кавычки.

unused constant: \* *(Синтаксический анализатор)*

(неиспользуемая константа)

Эта константа никогда не используется. Может быть, она не нужна совсем.

unused enum: \* *(Синтаксический анализатор)*

(неиспользуемое перечисление)

Этот перечислимый тип никогда не используется. Возможно, он не нужен вообще.

unused label: \* *(Синтаксический анализатор)*

(неиспользуемая метка)

Эта метка никогда не используется. Может быть, она не нужна совсем.

unused member: \* *(Синтаксический анализатор)*

(неиспользованный член)

Этот член структуры никогда не используется. Может быть, он не нужен совсем.

unused structure: \* *(Синтаксический анализатор)*

(неиспользованная структура)

Эта структура никогда не используется. Может быть, она не нужна совсем.

unused typedef: \* *(Синтаксический анализатор)*

(неиспользованное определение типа: \*)

Это определение типа никогда не используется. Может быть, оно не нужно совсем.

unused union: \* *(Синтаксический анализатор)*

(неиспользуемое объединение: \*)

Это объединение никогда не используется. Может быть, оно не нужно совсем.

unused variable declaration: \* *(Синтаксический анализатор)*

(неиспользуемое объявление переменной: \*)

Эта переменная никогда не используется. Может быть, она не нужна совсем.

unused variable definition: \* *(Синтаксический анализатор)*

(неиспользуемое определение переменной: \*)

Эта переменная никогда не используется. Может быть, она не нужна совсем.

upper case #include files are non-portable *(Препроцессор)*

(Файлы #include в верхнем регистре непереносимы)

При использовании DOS не имеет значения регистр файла #include. В других операционных системах регистр имеет значение.

variable may be used before set: \* *(Генератор кода)*

(переменная может быть использована перед установкой: \*)

Эта переменная может использоваться до назначения значения. Поскольку это автоматическая переменная, это приведет к тому, что она будет иметь случайное значение.

void function cannot return value *(Синтаксический анализатор)*

(функция void не может вернуть значение)

Функция void не может вернуть значение. Любой оператор "return" не должен сопровождаться выражением.

while expected *(Синтаксический анализатор)*

(ожидается while)

В конце оператора "do" ожидается ключевое слово "while".

work buffer overflow doing \* ## *(Препроцессор)*

(переполнение рабочего буфера при выполнении \* ##)

Это внутренняя ошибка компилятора. Обратитесь в службу технической поддержки HI-TECH Software за подробной информацией.

work buffer overflow: \* *(Препроцессор)*

(переполнение рабочего буфера: \*)

Это внутренняя ошибка компилятора. Обратитесь в службу технической поддержки HI-TECH Software за подробной информацией.

write error (out of disk space?) \* *(Компоновщик)*

(Ошибка записи (недостаточно места на диске?))

Вероятно, жесткий диск переполнен.

write error on \* *(Ассемблер, Компоновщик, Cromwell)*

(Ошибка записи в \*)

В указанном файле произошла ошибка записи. Вероятно, это означает, что вы исчерпали дисковое пространство.

write error on object file *(Ассемблер)*

(Ошибка записи в объектный файл)

При попытке ассемблера записать объектный файл сообщается об ошибке. Вероятно, это означает, что на диске недостаточно свободного места.

wrong number of macro arguments for \* - \* instead of \* *(Препроцессор)*

(неправильное число аргументов макроса для \* - \* вместо \*)

Макрос был вызван с неправильным количеством аргументов.

# Библиотечные функции

В этой главе перечислены функции, входящие в библиотеку компилятора Hi-Tech C Z80/Z180. Каждая запись начинается с имени функции. После этого информация анализируется в следующих разделах.

**Синтаксис**

Это определение функции и файл заголовка, в котором она объявлена.

**Описание**

Это описание функции и ее назначение.

**Пример**

Это пример использования функции. Это, как правило, полная небольшая программа, которая иллюстрирует функцию.

**Типы данных**

Если какие-либо специальные типы данных (структуры и т. д.) определены для использования с функцией, они перечислены здесь с определением C. Эти типы данных будут определены в заголовочном файле, указанном под заголовком Синтаксис.

**См также**

Это относится к любым смежным функциям.

**Возвращаемое значение**

Указывается тип и характер возвращаемого значения функции, если такое имеется. Также включена информация об возврате ошибок.

Используются только те заголовки, которые имеют отношение к каждой функции.

**ABORT**

***Синтаксис:***

**#include <stdlib.h>**

**void abort (void)**

***Описание:***

Функция **abort()** вызывается для ненормального завершения работы программы. Она напечатает соответствующее сообщение и выйдет с отрицательным статусом (-1), только CP/M. В DOS пользователю необходимо предоставить инструкцию printf перед вызовом abort. Эта функция недоступна для встроенных программ.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

void main (void) {

char \* ptr, c;

ptr = &c;

if(ptr == NULL) abort();

}

***Возвращаемое значение:***

Отсутствуют

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**ABS**

***Синтаксис:***

**#include <stdlib.h>**

**int abs (int j)**

***Описание:***

Функция **abs()** возвращает абсолютное значение **j**.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

void main (void) {

int a = -5;

printf("The absolute value of %d is %d\n", a, abs(a));

}

***Возвращаемое значение:***

Абсолютное значение **j**.

**ACOS**

***Синтаксис:***

**#include <math.h>**

**double acos (double f)**

***Описание:***

Функция **acos()** реализует обратное значение cos(), то есть передается значение в диапазоне от -1 до +1 и возвращает угол в радианах, косинус которых равен этому значению.

***Пример:***

#include <math.h>

#include <stdio.h>

/\* Печать значения acos() от -1 до 1 в градусах. \*/

void main (void) {

float i, a;

for(i = -1.0; i < 1.0 ; i += 0.1) {

a = acos(i)\*180.0/3.141592;

printf("acos(%f) = %f degrees\n", i, a);

}

}

***См. также***

[sin()](#f_sin), [cos()](#f_cos), [tan()](#f_tan), [asin()](#f_asin), [atan()](#f_atan), [atan2()](#f_atan2)

***Возвращаемое значение:***

Угол в радианах в диапазоне от 0 до π. Если значение аргумента находится вне диапазона от -1 до 1, возвращаемое значение будет равно нулю.

**ASCTIME**

***Синтаксис:***

**#include <time.h>**

**char \* asctime (struct tm \* t)**

***Описание:***

Функция **asctime()** принимает представление времени в виде **struct tm**, на которое указывает ее параметр **t**, и возвращает строку из 26 символов, описывающих текущую дату и время в формате:

Sun Sep 16 01:03:52 1973\n\0

Обратите внимание на символ новой строки в конце строки. Ширина каждого поля в строке фиксирована. Пример получает текущее время, преобразует его в указатель **struct tm** с помощью функции localtime(), затем преобразует его в ASCII и печатает его. Функция time() должна быть предоставлена пользователем (см. описание [time()](#f_time) для получения дополнительной информации).

***Пример:***

#include <stdio.h>

#include <time.h>

void main (void) {

time\_t clock;

struct tm \* tp;

time(&clock);

tp = localtime(&clock);

printf("%s", asctime(tp));

}

***См. также***

[ctime()](#f_ctime), [gmtime()](#f_gmtime), [localtime()](#f_localtime), [time()](#f_time)

***Возвращаемое значение:***

Указатель на строку.

***Примечание:***

В примере требуется, чтобы пользователь предоставил процедуру time(), поскольку она не может быть предоставлена компилятором. См. описание [time()](#f_time) для получения дополнительной информации.

***Типы данных:***

struct tm {

int tm\_sec;

int tm\_min;

int tm\_hour;

int tm\_mday;

int tm\_mon;

int tm\_year;

int tm\_wday;

int tm\_yday;

int tm\_isdst;

};

**ASIN**

***Синтаксис:***

**#include <math.h>**

**double asin (double f)**

***Описание:***

Функция **asin()** реализует функцию обратную sin(), т. е. ей передается значение в диапазоне от -1 до +1, и возвращает угол в радианах, синус которого равен этому значению.

***Пример:***

#include <math.h>

#include <stdio.h>

void main (void) {

float i, a;

for(i = -1.0; i < 1.0 ; i += 0.1) {

a = asin(i)\*180.0/3.141592;

printf("asin(%f) = %f degrees\n", i, a);

}

}

***См. также***

[sin()](#f_sin), [cos()](#f_cos), [tan()](#f_tan), [acos()](#f_acos), [atan()](#f_atan), [atan2()](#f_atan2)

***Возвращаемое значение:***

Угол в радианах, в диапазоне от -π/2 до + π/2. Если значение аргумента находится за пределами диапазона от -1 до +1, возвращаемое значение будет равно нулю.

**ASSERT**

***Синтаксис:***

**#include <assert.h>**

**void assert (int e)**

***Описание:***

Этот макрос используется для отладки. Основной метод использования заключается в свободном размещении утверждений по всему коду в точках, где правильная работа кода зависит от определенных условий, которые изначально выполняются. Процедура **assert()** может использоваться, чтобы гарантировать во время выполнения, что предположение имеет значение true. Например, следующая инструкция утверждает, что указатель **tp** не равен NULL:

**assert(tp);**

Если во время выполнения выражение имеет значение false, программа прерывает работу с сообщением, идентифицирующим исходный файл и номер строки утверждения, и выражение, используемое в качестве аргумента. Более полное обсуждение использования **assert()** невозможно в ограниченном пространстве, но тесно связано с методами проверки правильности программ.

***Пример:***

void ptrfunc (struct xyz \* tp) {

assert(tp != 0);

}

***Примечание:***

При необходимости для систем на основе ПЗУ, базовая подпрограмма \_fassert(...) должна быть реализована пользователем.

**ATAN**

***Синтаксис:***

**#include <math.h>**

**double atan (double x)**

***Описание:***

Эта функция возвращает арктангенс аргумента, т. е. она возвращает угол в диапазоне от -π/2 до +π/2 такой, что tan(e) == x.

***Пример:***

#include <stdio.h>

#include <math.h>

void main (void) {

printf("%f\n", atan(1.5));

}

***См. также***

[sin()](#f_sin), [cos()](#f_cos), [tan()](#f_tan), [asin()](#f_asin), [acos()](#f_acos), [atan2()](#f_atan2)

***Возвращаемое значение:***

Арктангенс аргумента.

**ATAN2**

***Синтаксис:***

**#include <math.h>**

**double atan2 (double y, double x)**

***Описание:***

Эта функция возвращает арктангенс **y**/**x**, используя знак обоих аргументов для определения квадранта возвращаемого значения.

***Пример:***

#include <stdio.h>

#include <math.h>

void main (void) {

printf("%f\n", atan2(1.5, 1));

}

***См. также***

[sin()](#f_sin), [cos()](#f_cos), [tan()](#f_tan), [asin()](#f_asin), [acos()](#f_acos), [atan()](#f_atan)

***Возвращаемое значение:***

Арктангенс y/x в диапазоне от -π до +π радиан. Если и y и x равны нулю, возникает доменная ошибка и возвращается ноль.

**ATEXIT**

***Синтаксис:***

**#include <stdlib.h>**

**int atexit (void (\*func)(void))**

***Описание:***

Функция **atexit()** регистрирует функцию, на которую указывает **func**, которая вызывается без аргументов при нормальном завершении программы. При завершении программы все функции, зарегистрированные **atexit()**, вызывают в обратном порядке их регистрации.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

char \* fname;

FILE \* fp;

void rmfile (void) {

if(fp)

close(fp);

if(fname)

remove(fname);

}

/\* создать файл; при выходе закрыть и удалить его \*/

void main (void) {

if(!(fp = fopen((fname = "test.fil"), "w")))

exit(1);

atexit(rmfile);

}

***См. также***

[exit()](#f_exit)

***Возвращаемое значение:***

Функция **atexit()** возвращает ноль, если регистрация завершается успешно, ненулевое значение в случае сбоя.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**ATOF**

***Синтаксис:***

**#include <stdlib.h>**

**double atof (const char \* s)**

***Описание:***

Функция **atof()** сканирует переданную ему строку символов, пропуская ведущие пробелы. Затем она преобразует ASCII-представление числа в double. Число может быть в десятичной, нормальной плавающей запятой или научной нотации.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

void main (void) {

char buf[80];

double i;

gets(buf);

i = atof(buf);

printf("Read %s: converted to %f\n", buf, i);

}

***См. также***

[atoi()](#f_atoi), [atol()](#f_atol)

***Возвращаемое значение:***

Число двойной точности с плавающей запятой. Если число в строке не будет найдено, то будет возвращен 0.0.

**ATOI**

***Синтаксис:***

**#include <stdlib.h>**

**double atoi (const char \* s)**

***Описание:***

Функция **atoi()** проверяет переданную ему строку символов, пропускает предшествующие пробелы и считывает необязательный знак. Затем она преобразует ASCII-представление десятичного числа в целое.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

void main (void) {

char buf[80];

int i;

gets(buf);

i = atoi(buf);

printf("Read %s: converted to %d\n", buf, i);

}

***См. также***

[xtoi()](#f_xtoi), [atof()](#f_atof), [atol()](#f_atoi)

***Возвращаемое значение:***

Целое число со знаком. Если в строке число не найдено, будет возвращен 0.

**ATOL**

***Синтаксис:***

**#include <stdlib.h>**

**long atol (const char \* s)**

***Описание:***

Функция **atol()** сканирует переданную ей строку символов, пропуская начальные пробелы. Затем выполняется преобразование представление в формате ASCII десятичного числа в длинное целое число.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

void main (void) {

char buf[80];

long i;

gets(buf);

i = atol(buf);

printf("Read %s: converted to %ld\n", buf, i);

}

***См. также***

[atoi()](#f_atoi), [atof()](#f_atof)

***Возвращаемое значение:***

Длинное целое число. Если число в строке не будет найдено, то будет возвращен 0.0.

**BDOS**

***Синтаксис:***

**#include <cpm.h>**

**char bdos (int func, int arg)**

***Описание:***

Функция **bdos()** вызывает CP/M BDOS с **func** в регистре C (CL для CP/M-86) и **arg** в регистре DE (DX), если требуется. Возвращаемое значение - байт, возвращаемый BDOS в регистре A (AX). Константы для различных значений функции BDOS определяются в **cpm.h**.

***Пример:***

#include <cpm.h>

/\* проверить, была ли нажата клавиша \*/

char kbhit (void) {

return(bdos(0x0B) & 0xFF) != 0;

}

***См. также***

[bdoshl()](#f_bdoshl), [bios()](#f_bios), msdos()

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**BDOSHL**

***Синтаксис:***

**#include <cpm.h>**

**short bdoshl (int, ...)**

***Описание:***

Эта функция выполнит вызов BDOS CP/M, возвращая значение, содержавшееся в HL после вызова. Она используется для некоторых вызовов BDOS, где возвращаемое значение находится в HL. Пример показывает использование **bdoshl()**, для определения номера версии CP/M.

***Пример:***

#include <cpm.h>

#include <stdio.h>

void main (void) {

short verno;

verno = bdoshl(12);

printf("CP/M version number %d.%d\n", (verno>>4)&0xF, verno & 0xF);

}

***См. также***

[bdos()](#f_bdos), [bios()](#f_bios), msdos()

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**BIOS**

***Синтаксис:***

**#include <cpm.h>**

**char bios (int n, int a1, int a2)**

***Описание:***

Эта функция вызовет **n**-ую точку входа BIOS (холодная загрузка = 0, теплая загрузки = 1, и т. д.) с регистром BC (CX), установленным из аргумента **a1** и DE (DX), установленным из аргумента **a2**. Возвращаемое значение - содержимое регистра A (AX) после вызова BIOS. В CP/M-86 функция 50 BDOS используется для выполнения вызова BIOS. Эта функция не должна использоваться, если возможно, так как она непереносима.. Даже не гарантируется переносимость вызовов BIOS между разными системами CP/M.

***См. также***

[bdos()](#f_bdos)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**BSEARCH**

***Синтаксис:***

**#include <stdlib.h>**

**void \* bsearch (const void \* key, void \* base, size\_t n\_memb,**

**size\_t size, int (\*compar)(const void \*, const void \*))**

***Описание:***

Функция **bsearch()** выполняет поиск элемента в отсортированном массиве, соответствующего определенному ключу **key**. Она использует алгоритм двоичного поиска, вызывая функцию для сравнения элементов в массиве, на которую указывает **compar**.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

struct value {

char name[40];

int value;

} values[100];

int val\_cmp (const void \* p1, const void \* p2) {

return strcmp(((const struct value \*)p1)->name,

((const struct value \*)p2)->name);

}

void main (void) {

char inbuf[80];

int i;

struct value \* vp;

i = 0;

while(gets(inbuf)) {

sscanf(inbuf,"%s %d", values[i].name, &values[i].value);

i++;

}

qsort(values, i, sizeof values[0], val\_cmp);

vp = bsearch("fred", values, i, sizeof values[0], val\_cmp);

if(!vp)

printf("Item ’fred’ was not found\n");

else

printf("Item ’fred’ has value %d\n", vp->value);

}

***См. также***

[qsort()](#f_qsort)

***Возвращаемое значение:***

Указатель на соответствующий элемент массива (если имеется несколько совпадающих элементов, может быть возвращен любой из них). Если совпадение не найдено, возвращается пустой указатель.

***Примечание:***

Функция сравнения должна иметь корректный прототип.

**CALLOC**

***Синтаксис:***

**#include <stdlib.h>**

**void \* calloc (size\_t cnt, size\_t size)**

***Описание:***

Функция **calloc()** пытается получить непрерывный блок динамической памяти, который будет содержать **cnt** объектов, каждый длиной **size**. Блок заполняется нулями.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

struct test {

int a[20];

} \* ptr;

/\* Выделение пространства для 20 структур. \*/

void main (void) {

ptr = calloc(20, sizeof(struct test));

if(!ptr)

printf("Failed\n");

else

free(ptr);

}

***См. также***

brk(), [sbrk()](#f_sbrk), [malloc()](#f_malloc), [free()](#f_free)

***Возвращаемое значение:***

Возвращается указатель на блок или нуль, если память не может быть выделена.

**CEIL**

***Синтаксис:***

**#include <math.h>**

**double ceil (double f)**

***Описание:***

Эта процедура возвращает наименьшее целое число не меньше, чем **f**.

***Пример:***

#include <stdlib.h>

#include <math.h>

void main (void) {

double j;

scanf("%lf", &j);

printf("The ceiling of %lf is %lf\n", j, ceil(j));

}

**CGETS**

***Синтаксис:***

**#include <conio.h>**

**char \* cgets (char \* s)**

***Описание:***

Функция **cgets()** считывает одну строку ввода с консоли в буфер, переданный в качестве аргумента. Это делается путем повторных вызовов getche(). При чтении символов они заносятся в буфер, backspace удаляет ранее введенный символ, а ctrl-U удаляет всю строку, напечатанную до сих пор. Другие символы помещаются в буфер, с символами возврата каретки или перевода строки (новая строка), завершающих функцию. Собранная строка завершается нулем.

***Пример:***

#include <conio.h>

#include <string.h>

char buffer[80];

void main (void) {

for(;;) {

cgets(buffer);

if(strcmp(buffer, "exit") == 0)

break;

cputs("Type ’exit’ to finish\n");

}

}

***См. также***

[getch()](#f_getch), [getche()](#f_getche), [putch()](#f_putch), [cputs()](#f_cputs)

***Возвращаемое значение:***

Возвращаемым значением является указатель на символ, передаваемый в качестве единственного аргумента.

**CHMOD**

***Синтаксис:***

**#include <stat.h>**

**#include <unixio.h>**

**int chmod (const char \* name, unsigned mode)**

***Описание:***

Эта функция изменяет атрибуты файла (или режимы) названного файла. Аргумент **name** может быть любым допустимым именем файла. Аргумент **mode** может включать все биты, определенные в **stat.h** кроме тех, которые относятся к типу файла, например S\_IFDIR.

***Пример:***

#include <stat.h>

#include <stdio.h>

#include <unixio.h>

/\* сделать файл только для чтения \*/

void main (int argc, char \*\* argv) {

if(argc > 1)

if(chmod(argv[1], S\_IREAD) < 0)

perror(argv[1]);

}

***См. также***

[stat()](#f_stat), [creat()](#f_creat)

***Возвращаемое значение:***

Нуль возвращается при успехе, минус единица (-1) при отказе.

***Примечание:***

Не все биты могут быть изменены во всех операционных системах, например, ни DOS, ни CP/M не позволяют сделать файл нечитаемым, таким образом, даже если режим не включает S\_IREAD файл по-прежнему будет читаемым (и stat() все еще будет возвращать в флагах S\_IREAD).

Эта процедура не может использоваться в системе на основе ПЗУ.

**CLOSE**

***Синтаксис:***

**#include <unixio.h>**

**int close (int fd)**

***Описание:***

Эта подпрограмма закрывает файл, связанный с дескриптором файла **fd**, который был ранее получен с помощью вызова функции open() или creat().

***Пример:***

#include <unixio.h>

#include <stdio.h>

/\* создание пустого файла \*/

void main (int argc, char \*\* argv) {

int fd;

if(argc > 1) {

if((fd = creat(argv[1], 0600)) < 0)

perror(argv[1]);

else

close(fd);

}

}

***См. также***

[open()](#f_open), [read()](#f_read), [write()](#f_write), seek()

***Возвращаемое значение:***

Возвращает ноль для успешного закрытия или отрицательный (-1) в противном случае.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**CLRERR, CLREOF**

***Синтаксис:***

**#include <stdio.h>**

**void clrerr (FILE \* stream)**

**void clreof (FILE \* stream)**

***Описание:***

Это макросы, определенные в **stdio.h**, которые сбрасывают ошибку и флаг конца файла, соответственно. для указанного потока **stream**. Они должны использоваться осторожно. Основное допустимое использование заключается в очистке состояния конца файла при вводе с терминального устройства для продолжения чтения после просмотра конца файла. Если **clreof()** не выполняется, повторные чтения будут продолжать возвращать EOF.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

char buf[80];

for(;;) {

if(!gets(buf)) {

printf("EOF seen\n");

clreof(stdin);

} else if(strcmp(buf, "quit") == 0)

break;

}

}

***См. также***

[fopen()](#f_fopen), [fclose()](#f_fclose)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**COS**

***Синтаксис:***

**#include <math.h>**

**double cos (double f)**

***Описание:***

Эта функция возвращает косинус своего аргумента, который является углом в радианах. Косинус вычисляется путем расширения аппроксимации ряда полиномов.

***Пример:***

#include <math.h>

#include <stdio.h>

#define C 3.141592/180.0

void main (void) {

double i;

for(i = 0 ; i <= 180.0 ; i += 10)

printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i\*C), cos(i\*C));

}

***См. также***

[sin()](#f_sin), [tan()](#f_tan), [asin()](#f_asin), [acos()](#f_cos), [atan()](#f_atan), [atan2()](#f_atan2)

***Возвращаемое значение:***

Значение двойной точности в диапазоне от -1 до +1.

**COSH, SINH, TANH**

***Синтаксис:***

**#include <math.h>**

**double cosh (double f)**

**double sinh (double f)**

**double tanh (double f)**

***Описание:***

Эти функции - гиперболические реализации тригонометрических функций: cos(), sin() и tan().

***Пример:***

#include <stdio.h>

#include <math.h>

void main (void) {

printf("%f\n", cosh(1.5));

printf("%f\n", sinh(1.5));

printf("%f\n", tanh(1.5));

}

***Возвращаемое значение:***

Функция **cosh()** возвращает значение гиперболического косинуса.

Функция **sinh()** возвращает гиперболическое значение синуса.

Функция **tanh()** возвращает гиперболическое значение тангенса.

**CPUTS**

***Синтаксис:***

**#include <conio.h>**

**void cputs (const char \* s)**

***Описание:***

Функция **cputs()** записывает строку аргумента в консоль, выводит возврат каретки перед каждой новой строкой в строке. Он многократно вызывает putch(). В компьютерной системе **cputs()** отличается от puts() тем, что она читает консоль напрямую, а не использует файловый ввод-вывод. Во встроенной системе **cputs()** и puts() эквивалентны.

***Пример:***

#include <conio.h>

#include <string.h>

char buffer[80];

void main (void) {

for(;;) {

cgets(buffer);

if(strcmp(buffer, "exit") == 0)

break;

cputs("Type ’exit’ to finish\n");

}

}

***См. также***

[cputs()](#f_cputs), [puts()](#f_puts), [putch()](#f_putch)

**CREAT**

***Синтаксис:***

**#include <stat.h>**

**#include <unixio.h>**

**int creat (const char \* name, int mode)**

***Описание:***

Эта подпрограмма пытается создать файл с именем **name**. Если файл существует и доступен для записи, он будет удален и создан заново. Возвращаемое значение минус единица (-1), если создание не выполнено, или небольшое неотрицательное число, если оно выполнено. Это число является ценным маркером, который должен использоваться для записи или закрытия файла впоследствии. Параметр **mode** используется для инициализации атрибутов создаваемого файла. Допустимые биты те же, что и для chmod(), но для совместимости с Unix рекомендуется использовать режим 0666 или 0600. В CP/M режим игнорируется - единственный способ установить атрибуты файла – с помощью функции chmod().

***Пример:***

#include <unixio.h>

#include <stat.h>

#include <stdio.h>

void main (int argc, char \*\* argv) {

int fd;

if(argc > 1) {

if((fd = creat(argv[1], 0600)) < 0)

perror(argv[1]);

else

close(fd);

}

}

***См. также***

[open()](#f_close), [close()](#f_close), [read()](#f_read), [write()](#f_write), seek(), [stat()](#f_stat), [chmod()](#f_chmod)

***Возвращаемое значение:***

В случае успеха она вернет небольшое положительное число. При неудаче возвращается минус единица (-1).

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**CTIME**

***Синтаксис:***

**#include <time.h>**

**char \* ctime (time\_t \* t)**

***Описание:***

Функция **ctime()** преобразует время в секундах, на которое указывает его аргумент, в строку той же формы, что и для asctime(). Таким образом, пример программы печатает текущее время и дату.

***Пример:***

#include <stdio.h>

#include <time.h>

void main (void) {

time\_t clock;

time(&clock);

printf("%s", ctime(&clock));

}

***См. также***

[gmtime()](#f_gmtime), [localtime()](#f_localtime), [asctime()](#f_astime), [time()](#f_time)

***Возвращаемое значение:***

Указатель на строку.

***Примечание:***

В примере требуется, чтобы пользователь предоставил процедуру time(), поскольку она не может быть предоставлена компилятором. См. описание [time()](#f_time) для получения дополнительной информации.

***Типы данных:***

typedef long time\_t;

**DI, EI**

***Синтаксис:***

**#include <intrpt.h>**

**void ei (void)**

**void di (void)**

***Описание:***

Процедуры **ei()** и **di()** позволяют включать и отключать прерывания соответственно. Они реализованы как макросы, определенные в **intrpt.h**. На большинстве процессоров они будут расширяться до встроенной инструкции ассемблера, которая устанавливает или очищает бит включения или маски прерывания.

В этом примере показано использование **ei()** и **di()** в момент доступа к длинной переменной, которая изменяется во время прерывания. Если бы это не было сделано, было бы возможно вернуть неправильное значение, если бы прерывание произошло между доступами к последовательным словам значения count.

***Пример:***

#include <intrpt.h>

long count;

void interrupt tick (void) {

count++;

}

long getticks (void) {

long val;

di(); /\* отключение прерываний вокруг доступа к \*/

val = count; /\* count для обеспечения корректности \*/

ei();

return val;

}

**DIV**

***Синтаксис:***

**#include <stdlib.h>**

**div\_t div (int numer, int demon)**

***Описание:***

Функция **div()** вычисляет частное и остаток числителя, деленного на знаменатель.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

void main (void) {

div\_t x;

x = div(12345, 66);

printf("quotient = %d, remainder = %d\n", x.quot, x.rem);

}

***Возвращаемое значение:***

Возвращает частное и остаток в структуру **div\_t**.

***Типы данных:***

typedef struct {

int quot;

int rem;

} div\_t;

**DUP**

***Синтаксис:***

**#include <unixio.h>**

**int dup (int fd)**

***Описание:***

Учитывая дескриптор файла, например, возвращаемый функцией open(), эта процедура возвращает другой дескриптор файла, который будет ссылаться на тот же открытый файл.

***Пример:***

#include <stdio.h>

#include <unixio.h>

#include <stdlib.h>

#include <sys.h>

void main (int argc, char \*\* argv) {

FILE \* fp;

if(argc < 3) {

fprintf(stderr, "Usage: fd2 stderr\_file command ...\n");

exit(1);

}

if(!(fp = fopen(argv[1], "w"))) {

perror(argv[1]);

exit(1);

}

close(2);

dup(fileno(fp)); /\* создать ссылку на файл stderr \*/

fclose(fp);

spawnvp(argv[2], argv+2);

close(2);

}

***См. также***

[open()](#f_open), [close()](#f_close), [creat()](#f_creat), [read()](#f_read), [write()](#f_write)

***Возвращаемое значение:***

Возвращается минус единица (-1), если параметр **fd** является плохим дескриптором или не относится к открытому файлу.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**EVAL\_POLY**

***Синтаксис:***

**#include <math.h>**

**double eval\_poly (double x, const double \* d, int n)**

***Описание:***

Функция **eval\_poly()** вычисляет полином, коэффициенты которого содержатся в массиве **d** в **x**, например:

y = x\*x\*d2 + x\*d1 + d0

Порядок полинома передается в **n**.

***Пример:***

#include <stdlib.h>

#include <math.h>

void main (void) {

double x, y;

double d[3] = {1.1, 3.5, 2.7};

x = 2.2;

y = eval\_poly(x, d, 2);

printf("The polynomial evaluated at %f is %f\n", x, y);

}

***Возвращаемое значение:***

Значение типа double, являющееся полиномом, вычисленным в **x**.

**EXECL, EXECV**

***Синтаксис:***

**#include <sys.h>**

**int execl (char \* name, pname, ...)**

**int execv (char \* name, ppname)**

***Описание:***

Функции загружают и выполняют программу, определенную именем строкой **name**. Процедура **execl()** принимает аргументы для программы из строки списка аргументов, завершающейся нулем. Функции **execv()** передается указатель на массив строк. Массив должен быть завершен нулем. Если названная программа найдена и может быть считана, вызов не возвращается. Таким образом любой возврат из этих подпрограмм можно рассматривать как ошибку.

***Пример:***

#include <cpm.h>

#include <stdio.h>

#include <sys.h>

void main (void) {

execl("tst.com", "tst", "argument 1", 0);

perror("tst.com");

}

***См. также***

spawnl(), spawnv(), system()

***Примечание:***

Второй аргумент **execl()** или первая строка в массиве строк, переданных **execv()**, номинально argv[0] в выполняемой программе. Однако, поскольку CP/M не может передать имя программы, эта строка будет неиспользуемой. Однако она должна присутствовать в качестве заполнителя.

Эта процедура не может использоваться в системе на основе ПЗУ.

**EXIT**

***Синтаксис:***

**#include <stdlib.h>**

**void exit (int status)**

***Описание:***

Этот вызов закроет все открытые файлы и выход из программы. В CP/M это значит возврат на уровень CCP, в MS-DOS возврат к командной строке DOS или программе, которая породила текущую программу. Значение **status** используется в качестве значения выхода программы. Он восстанавливается в DOS с ожиданием вызова состояния DOS. Значение **status** в CP/M сохраняется по адресу в 80H. Во встроенной системе **exit()** обычно перезапускает программу, как если бы произошел аппаратный сброс. Этот вызов никогда не осуществляет возврат.

***Пример:***

#include < stdlib.h>

void main (void) {

exit(0);

}

***См. также***

[atexit()](#f_atexit)

***Возвращаемое значение:***

Отсутствует.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**EXP**

***Синтаксис:***

**#include <math.h>**

**double exp (double f)**

***Описание:***

Подпрограмма **exp()** возвращает экспоненциальную функцию своего аргумента, т.е. e в степени **f**.

***Пример:***

#include <math.h>

#include <stdio.h>

void main (void) {

double f;

for(f = 0.0 ; f <= 5 ; f += 1.0)

printf("e to %1.0f = %f\n", f, exp(f));

}

***См. также***

[log()](#f_log), [log10()](#f_log10), [pow()](#f_pow)

**FABS**

***Синтаксис:***

**#include <math.h>**

**double fabs (double f)**

***Описание:***

Эта процедура возвращает абсолютное значение его двойного аргумента.

***Пример:***

#include <stdio.h>

#include <math.h>

void main (void) {

printf("%f %f\n", fabs(1.5), fabs(-1.5));

}

***См. также***

[abs()](#f_abs)

**FCLOSE**

***Синтаксис:***

**#include <stdio.h>**

**int fclose (FILE \* stream)**

***Описание:***

Эта процедура закрывает указанный поток ввода-вывода. Значение **stream** должен быть файловым дескриптором, возвращенным предыдущим вызовом fopen().

***Пример:***

#include <stdio.h>

void main (int argc, char \*\* argv) {

FILE \* fp;

if(argc > 1) {

if(!(fp = fopen(argv[1], "r")))

perror(argv[1]);

else {

fprintf(stderr, "Opened %s\n", argv[1]);

fclose(fp);

}

}

}

***См. также***

[fopen()](#f_fopen), [fread()](#f_fread), [fwrite()](#f_fwrite)

***Возвращаемое значение:***

При успешном завершении возвращается ноль, иначе EOF.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FDOPEN**

***Синтаксис:***

**#include <stdio.h>**

**FILE \* fdopen (int fd, const char \* mode)**

***Описание:***

Эта процедура закрывает указанный поток ввода-вывода. Значение потока должно быть файловым дескриптором, возвращенным предыдущим вызовом fopen().

***Пример:***

#include <stdio.h>

void main (void) {

FILE \* fp;

/\* 3 это устройство AUX \*/

fp = fdopen(3, "w");

fprintf(fp, "AUX test line\n");

}

***См. также***

[fopen()](#f_fopen), [freopen()](#f_freopen), [close()](#f_close)

***Возвращаемое значение:***

Возвращается NULL, если структура FILE не может быть выделена.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FEOF, FERROR**

***Синтаксис:***

**#include <stdio.h>**

**int feof (FILE \* stream)**

**int ferror (FILE \* stream)**

***Описание:***

Эти макросы проверяют состояние битов конца файла и ошибок соответственно для указанного потока. Каждый из них будет true, если установлен соответствующий флаг. Макросы определяются в stdio.h. Значение **stream** должен быть файловым дескриптором, возвращаемым предыдущим вызовом fopen().

***Пример:***

#include <stdio.h>

void main (void) {

while(!feof(stdin))

getchar();

}

***См. также***

[fopen()](#f_fopen), [fclose()](#f_fclose)

***Возвращаемое значение:***

Функция **feof()** возвращает ненулевое значение, если для потока установлен индикатор конца файла.

Функция **ferror()** возвращает ненулевое значение, если индикатор ошибки установлен для потока.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FFLUSH**

***Синтаксис:***

**#include <stdio.h>**

**int fflush (FILE \* stream)**

***Описание:***

Функция **fflush()** выведет в файл на диске, или другое открытое в настоящее время устройство в указанном потоке **stream** содержимое связанного с ним буфера. Она обычно используется для сбрасывания буферизированного стандартного вывода в интерактивных приложениях. Обычно stdout открывается в режиме буферизации и очищается до того, как любой ввод будет выполнен в поток stdio, но если ввод будет осуществляться с помощью процедур ввода/вывода с консоли, то, возможно, сначала потребуется вызвать **fflush()**.

***Пример:***

#include <stdio.h>

#include <conio.h>

void main (void) {

printf("press a key: ");

fflush(stdout);

getch();

}

***См. также***

[fopen()](#f_fopen), [fclose()](#f_fclose)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FGETC**

***Синтаксис:***

**#include <stdio.h>**

**int fgetc (FILE \* stream)**

***Описание:***

Функция **fgetc()** возвращает следующий символ из входного потока. Если встречается конец файла, возвращается EOF. Именно по этой причине функция объявляется целочисленной. Целочисленный EOF не является допустимым байтом, поэтому конец файла отличается от байта, единицами во всех битах, прочитанного из файла. Подпрограмма **fgetc()** - это не-макро версия getc().

***Пример:***

#include <stdio.h>

void main (void) {

FILE \* fp;

int c;

fp = fopen( "file.dat", "r" );

if( fp != NULL ) {

while( (c = fgetc( fp )) != EOF )

fputc( c, stdout );

fclose( fp );

}

}

***См. также***

[fopen()](#f_fopen), [fclose()](#f_fclose), [fputc()](#f_fputc), [getc()](#f_getc), [putc()](#f_putc)

***Возвращаемое значение:***

Веденный символ из потока **stream** или EOF при обнаружении конца файла.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FGETS**

***Синтаксис:***

**#include <stdio.h>**

**char \* fgets (char \* s, int n, FILE \* stream)**

***Описание:***

Функция **fgets()** помещает в буфере **s** до **n-1** символов из входного потока **stream**. Если во вводе появится символ новой строки, прежде чем будет считано корректное число символов, то сразу произойдет возврат из **fgets()**. Символ новой строки останется в буфере. В любом случае буфер будет завершен нулем.

***Пример:***

#include <stdio.h>

void main (void) {

char buffer[128];

while(fgets(buffer, sizeof buffer, stdin))

fputs(buffer, stdout);

}

***См. также***

[fopen()](#f_fopen), [fclose()](#f_fclose), [fputc()](#f_fputc), [getc()](#f_getc), [putc()](#f_putc)

***Возвращаемое значение:***

Успешный вызов **fgets()** возвратит свой первый параметр. При достижении конца файла или ошибке возвращается NULL.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FILENO**

***Синтаксис:***

**#include <stdio.h>**

**int fileno (FILE \* stream)**

***Описание:***

Процедура **fileno()** является макросом из файла **stdio.h**, который возвращает дескриптор файла, связанный с потоком **stream**. Обычно, он используется, когда необходимо выполнить некоторую низкоуровневую операцию с файлом, открытым как поток stdio.

***Пример:***

#include <stdio.h>

#include <unixio.h>

#include <stdlib.h>

#include <sys.h>

void main (int argc, char \*\* argv) {

FILE \* fp;

if(argc < 3) {

fprintf(stderr, "Usage: fd2 stderr\_file command ...\n");

exit(1);

}

if(!(fp = fopen(argv[1], "w"))) {

perror(argv[1]);

exit(1);

}

close(2);

dup(fileno(fp)); /\* make stderr reference file \*/

fclose(fp);

spawnvp(argv[2], argv+2);

close(2);

}

***См. также***

[fopen()](#f_fopen), [fclose()](#f_fclose), [open()](#f_open), [close()](#f_close), [dup()](#f_dup)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FLOOR**

***Синтаксис:***

**#include <math.h>**

**double floor (double f)**

***Описание:***

Эта процедура возвращает наибольшее целое число не больше **f**.

***Пример:***

#include <stdio.h>

#include <math.h>

void main (void) {

printf("%f\n", floor( 1.5 ));

printf("%f\n", floor( -1.5));

}

**FOPEN**

***Синтаксис:***

**#include <stdio.h>**

**FILE \* fopen (const char \* name, const char \* mode)**

***Описание:***

Функция **fopen()** пытается открыть файл для чтения или записи (или того и другого) согласно предоставленной строки **mode**. Строка режима интерпретируется следующим образом:

**r** Файл открывается для чтения, если он существует. Если файл не существует, то вызов завершится с ошибкой.

**r+** Если файл существует, он открывается для чтения и записи. Если файл не существует, то вызов завершится с ошибкой.

**w** Файл создается, если он не существует, или усекается, если он есть. Затем он открывается для записи.

**w+** Файл создается, если он не существует, или усекается, если он есть. Файл открывается для чтения и записи.

**a** Файл создается, если он еще не существует, и открывается для записи. Все записи будут динамично принудительно приводить в конец файла, таким образом этот режим известен, как *режим добавления*.

**a+** Файл создается, если он еще не существует, и открывается для чтения и записи. Все записи в файл будут динамично приводить к концу файла, т.е. в то время как любая часть файла может быть считана, все записи будут располагаться в конце файла, и не перезапишут существующих данных. Вызов fseek() в попытке произвести запись в файл в любом другом месте не будет эффективным.

Модификатор "**b**" может быть добавлен к любому из вышеупомянутых режимов, например, "**r+b**" или "**rb+**" эквивалентны. Добавление модификатора "**b**" заставит файл быть открытым в двоичном, а не в ASCII режиме. Открытие в режиме ASCII гарантирует, что текстовые файлы считаны способом, совместимым с основанными на Unix соглашениями для программ C, то есть, что текстовые файлы содержат строки, разделенные символами новой строки. Специальный режим чтения или записи символов меняется в зависимости от операционной системы, но включает в себя некоторые или все из следующих действий:

**NEWLINE** (перевод строки), преобразуется в возврат каретки, перевод строки при выводе.

**RETURN** игнорируется при вводе, вставляется перед **NEWLINE** при выводе.

**Ctrl**-**Z** обозначает EOF при вводе. Добавляется fclose() при выводе при необходимости в CP/M.

Открытие файла в двоичном режиме позволяет каждый символ прочитать как он записан, но так как точный размер файла не известен CP/M, файл может содержать больше байтов, чем было записано в него. Смотрите open() для описание того, что является именем файла.

При использовании одного из режимов чтения/записи (с символом '+' в строке), несмотря на то, что они разрешают читать и писать в тот же поток, невозможно произвольно смешивать вызовы ввода и вывода в один и тот же поток. В любой момент времени поток, открытый в режиме "**+**", будет в состоянии ввода или вывода. Состояние может быть изменено, только когда связанный с ним буфер пуст, что гарантируется только сразу после вызова fflush() или одной из функций позиционирования файла fseek() или rewind(). Буфер также будет пуст после обнаружения EOF при чтении двоичного потока, но рекомендуется, использовать явный вызов, fflush(), чтобы гарантировать эту ситуацию. Таким образом, после чтения из потока вы должны вызвать fflush() или fseek() прежде, чем попытаться записать в этот поток, и наоборот.

***Пример:***

#include <stdio.h>

void main (int argc, char \*\* argv) {

FILE \* fp;

if(argc > 1) {

if(!(fp = fopen(argv[1], "r")))

perror(argv[1]);

else {

fprintf(stderr, "Opened %s\n", argv[1]);

fclose(fp);

}

}

}

***См. также***

[fclose()](#f_fclose), [fgetc()](#f_fgetc), [fputc()](#f_fputc), [freopen()](#f_freopen), [fread()](#f_fread), [fflush()](#f_fflush), [fwrite()](#f_fwrite), [fseek()](#f_fseek)

***Возвращаемое значение:***

Процедура **fopen()** возвращает указатель на объект файла. Если она завершается неудачей, возвращается нулевой указатель.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FPRINTF, VFPRINTF**

***Синтаксис:***

**#include <stdio.h>**

**int fprintf (FILE \* stream, const char \* fmt, ...)**

**#include <stdarg.h>**

**#include <stdio.h>**

**int vfprintf (FILE \* stream, const char \* fmt, va\_list va\_arg)**

***Описание:***

Функция **fprintf()** выполняет отформатированную печать в указанный поток **stream**. Обратитесь к **printf()** для получения подробной информации о доступных форматах. Функция **vfprintf()** похожа на **fprintf()**, но принимает указатель переменного списка аргументов, а не список аргументов. См. описание **va\_start()** для получения дополнительной информации о переменных списках аргументов.

***Пример:***

#include <stdio.h>

void main (int argc, char \*\* argv) {

FILE \* fp;

if(argc > 1) {

if(!(fp = fopen(argv[1], "r")))

perror(argv[1]);

else {

fprintf(stderr, "Opened %s\n", argv[1]);

fclose(fp);

}

}

}

***См. также***

[printf()](#f_printf), [fscanf()](#f_fscanf), [sscanf()](#f_sscanf), [sprintf()](#f_sprintf)

***Возвращаемое значение:***

Она возвращает число переданных символов или отрицательное число в случае неудачи.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FPUTC**

***Синтаксис:***

**#include <stdio.h>**

**int fputc (int c, FILE \* stream)**

***Описание:***

Символ **c** записывается в предоставленный поток **stream**. Это не макро версия putc().

***Пример:***

#include <stdio.h>

void main (void) {

FILE \* fp;

int c;

fp = fopen( "file.dat", "r" );

if( fp != NULL ) {

while( (c = fgetc( fp )) != EOF )

fputc( c, stdout );

fclose( fp );

}

}

***См. также***

[putc()](#f_putc), [fgetc()](#f_fgetc), [fopen()](#f_fopen), [fflush()](#f_fflush)

***Возвращаемое значение:***

Возвращается символ, если он был успешно записан, в противном случае возвращается EOF.

***Примечание:***

Обратите внимание, что "запись в поток" может означать только размещение символа в буфере, связанном с потоком.

Эта процедура не может использоваться в системе на основе ПЗУ.

**FPUTS**

***Синтаксис:***

**#include <stdio.h>**

**int fputs (const char \* s, FILE \* stream)**

***Описание:***

Строка **s**, заканчивающаяся нулем, записывается в поток **stream**. Новая строка не добавляется (см. puts()).

***Пример:***

#include <stdio.h>

void main (void) {

fputs("This is a line\n", stdout);

}

***См. также***

[puts()](#f_puts), [fgets()](#f_fgets), [fopen()](#f_fopen), [fclose()](#f_fclose)

***Возвращаемое значение:***

Возвращаемое значение равно нулю для успеха, EOF для ошибки.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FREAD**

***Синтаксис:***

**#include <stdio.h>**

**int fread (void \* buf, size\_t size, size\_t cnt, FILE \* stream)**

***Описание:***

Функция **fread()** читает до **cnt** объектов, каждый длиной **size**, в память в **buf** из потока **stream**. Выравнивание слов в потоке не предполагается или не требуется. Чтение выполняется с помощью последовательных вызовов getc().

***Пример:***

#include <stdio.h>

void main (void) {

char buf[80];

int i;

i = fread(buf, 1, sizeof(buf), stdin);

printf("Read %d bytes\n", i);

}

***См. также***

[fwrite()](#f_fwrite), [fopen()](#f_fopen), [fclose()](#f_fclose), [getc()](#f_getc)

***Возвращаемое значение:***

Возвращаемое значение - количество прочитанных объектов. Если ничего не прочитано, возвращается ноль. Обратите внимание, что возвращаемое значение меньше **cnt**, но больше нуля, может не представлять ошибку (см. описание [fwrite()](#f_fwrite)).

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FREE**

***Синтаксис:***

**#include <stdlib.h>**

**void free (void \* ptr)**

***Описание:***

Функция **free()** освобождает блок памяти с адресом **ptr**, который ранее должен был быть получен с помощью вызова malloc() или calloc().

***Пример:***

#include <stdlib.h>

#include <stdio.h>

struct test {

int a[20];

} \* ptr;

/\* Выделить место для 20 структур \*/

void main (void) {

ptr = calloc(20, sizeof(struct test));

if(!ptr)

printf("Failed\n");

else

free(ptr);

}

***См. также***

[malloc()](#f_malloc), [calloc()](#f_calloc)

**FREOPEN**

***Синтаксис:***

**#include <stdio.h>**

**FILE \* freopen (const char \* name, const char \* mode, FILE \* stream)**

***Описание:***

Функция **freopen()** закрывает заданный поток **stream** (если открыт), затем повторно открывает поток **stream**, прикрепленный к файлу с именем **name**. Режим открытия задается **mode**. Эта функция обычно используется для присоединения stdin или stdout к файлу, как в следующем примере.

***Пример:***

#include <stdio.h>

void main (void) {

char buf[80];

if(!freopen("test.fil", "r", stdin))

perror("test.fil");

if(gets(buf))

puts(buf);

}

***См. также***

[fopen()](#f_fopen), [fclose()](#f_fclose)

***Возвращаемое значение:***

Она возвращает либо аргумент потока, в случае успеха, либо NULL, если нет.

См. описание [fopen()](#f_fopen) для получения дополнительной информации.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FREXP**

***Синтаксис:***

**#include <math.h>**

**double frexp (double f, int \* p)**

***Описание:***

Функция **frexp()** раскладывает число с плавающей точкой на нормализованную дробную часть и целочисленную степень двойки. Целое число сохраняется в целочисленный объект, на который указывает **p**. Возвращаемое ей значение **x** находится в интервале (0.5, 1.0) или ноль, и **f** равняется значению **x** умноженному на 2 и возведенному в степень, хранящуюся в **\*p**. Если **f** - ноль, обе части результата равны нулю.

***Пример:***

#include <math.h>

#include <stdio.h>

void main (void) {

double f;

int i;

f = frexp(23456.34, &i);

printf("23456.34 = %f \* 2^%d\n", f, i);

}

***См. также***

[ldexp()](#f_ldexp)

**FSCANF, VFSCANF**

***Синтаксис:***

**#include <stdio.h>**

**int fscanf (FILE \* stream, const char \* fmt, ...)**

**#include <stdarg.h>**

**#include <stdio.h>**

**int vfscanf (FILE \* stream, const char \* fmt, va\_list va\_arg)**

***Описание:***

Эта подпрограмма выполняет форматированный ввод от указанного потока **stream**. См. описание scanf() для полного описания поведения подпрограммы.

Функция **vfscanf()** подобна **fscanf()**, но принимает указатель на переменный список, а не список аргументов. См. описание **va\_start()** для получения дополнительной информации о переменных списках аргументов.

***Пример:***

#include <stdio.h>

void main (void) {

int i;

printf("Enter a number: ");

fscanf(stdin, "%d", &i);

printf("Read %d\n", i);

}

***См. также***

[scanf()](#f_scanf), [sscanf()](#f_sscanf), [fopen()](#f_fopen), [fclose()](#f_fclose)

***Возвращаемое значение:***

Количество присвоенных значений или EOF, если произошла ошибка, и никакие элементы не были преобразованы.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FSEEK**

***Синтаксис:***

**#include <stdio.h>**

**int fseek (FILE \* stream, long offs, int wh)**

***Описание:***

Функция **fseek()** позиционирует "файловый указатель" (т.е. указатель на следующий символ, который будет считан или записан) указанного потока **stream** следующим образом:

**wh** Результирующее расположение

0 **offs** начало файла

1 **offs** + текущая позиция

2 **offs** + конец файла

Нужно отметить, что **offs** это значение со знаком. Таким образом, 3 предусмотренные режима обеспечивают позиционирование относительно начала файла, текущего указателя файла и конца файла соответственно. Однако, обратите внимание, что позиционирование за пределы конца файла разрешено, но приведет к индикации EOF, если предпринимается попытка считать там данные. Вполне допустимо, записать данные вне предшествующего конца файла. Функция **fseek()** правильно обрабатывает любые буферизированные данные. Текущее положение файла можно определить с помощью функции ftell().

***Пример:***

#include <stdio.h>

#include <stdlib.h>

void main (void) {

FILE \* fp;

/\* Открыть файл для чтения и записи\*/

fp = fopen("test.fil", "r+");

if(!fp)

exit(1);

fseek(fp,0L, 2); /\* стремиться к концу \*/

fputs("Another line!\n", fp);

fclose(fp);

}

***См. также***

[lseek()](#f_lseek), [fopen()](#f_fopen), [fclose()](#f_fclose), [ftell()](#f_ftell)

***Возвращаемое значение:***

Если запрос позиционирования не может быть удовлетворен возвращается EOF, в противном случае ноль.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FTELL**

***Синтаксис:***

**#include <stdio.h>**

**long ftell (FILE \* stream)**

***Описание:***

Эта функция возвращает текущее положение концептуального указателя чтения/записи, связанного с потоком **stream**. Это положение относительно начала файла следующего байта, который должен быть прочитан или записан в файл.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

void main (void) {

FILE \* fp;

fp = fopen("test.fil", "r");

if(!fp)

exit(1);

fseek(fp,0L, 2); /\* стремиться к концу \*/

printf("size = %ld\n", ftell(fp));

}

***См. также***

[fseek()](#f_fseek)

***Возвращаемое значение:***

Указатель на текущую позицию чтения/записи байта в указанном потоке **stream**.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**FWRITE**

***Синтаксис:***

**#include <stdio.h>**

**int fwrite (const void \* buf, size\_t size, size\_t cnt, FILE \* stream)**

***Описание:***

**cnt** объектов длиной **size** байтов будут записаны из памяти в **buf** из указанного потока **stream**.

***Пример:***

#include <stdio.h>

void main (void) {

if(fwrite("a test string\n", 1, 14, stdout) != 14)

fprintf(stderr, "Fwrite failed\n");

}

***См. также***

[fseek()](#f_fseek)

***Возвращаемое значение:***

Возвращается целочисленное количество записанных целых объектов или ноль, если ни один из них не будет записан. Любое возвращаемое значение, не равное **cnt**, должно рассматриваться как ошибка (см. описание [fread()](#f_fread)).

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**GETC**

***Синтаксис:***

**#include <stdio.h>**

**int getc (FILE \* stream)**

***Описание:***

Один символ считывается и возвращается из указанного потока **stream**. Это - макро-версия fgetc() и определена в файле **stdio.h**.

***Пример:***

#include <stdio.h>

void main (void) {

int i;

while((i = getc(stdin)) != EOF)

putchar(i);

}

***Возвращаемое значение:***

В конце файла или при ошибке возвращается EOF.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**GETCH, GETCHE**

***Синтаксис:***

**#include <conio.h>**

**char getch (void)**

**char getche (void)**

***Описание:***

Функция **getch()** читает один символ из консольной клавиатуры и возвращает его без эха. Функция **getche()** подобна, но повторяет введенный символ.

Во встроенной системе источник символов определяется заданными подпрограммами. По умолчанию библиотека содержит версию **getch()**, которая будет взаимодействовать через интерфейс с отладчиком Lucifer. Пользователь должен предоставить соответствующую процедуру, если требуется другой источник, например, последовательный порт.

Модуль getch.c в каталоге SOURCES содержит образцы версий всех подпрограмм консольного вода-вывода. Также могут предоставляться другие модули, например, ser180.c содержит процедуры последовательного порта Z180.

***Пример:***

#include <conio.h>

void main (void) {

char c;

while((c = getche()) != ’\n’)

continue;

}

***См. также***

[cgets()](#f_cgets), [cputs()](#f_cputs), [ungetch()](#f_ungetch)

**GETCHAR**

***Синтаксис:***

**#include <stdio.h>**

**int getchar (void)**

***Описание:***

Процедура **getchar()** является операцией getc(stdin). Это макрос, определенный в файле **stdio.h**. Обратите внимание, что при нормальных обстоятельствах **getchar()** не возвратится, пока в консоли не будет введен возврат каретки. Чтобы немедленно получить один символ с консоли, используйте подпрограмму getch().

***Пример:***

#include <stdio.h>

void main (void) {

int c;

while((c = getchar()) != EOF)

putchar(c);

}

***См. также***

[getc()](#f_getc), [fgetc()](#f_fgetc), [freopen()](#f_freopen), [fclose()](#f_close)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**GETENV**

***Синтаксис:***

**#include <stdlib.h>**

**char \* getenv (const char \* s)**

**extern char \*\* environ**

***Описание:***

Функция **getenv()** осуществляет поиск первого соответствия представленного параметра в таблице строк среды окружения и возвращает часть строки этой среды. Например, если среда содержит строку

COMSPEC=C:\COMMAND.COM

то getenv("COMSPEC") возвратит C:\COMMAND.COM. Глобальная переменная **environ** является указателем на массив указателей на строки среды, завершенный нулевым указателем. В MS-DOS этот массив инициализируется во время запуска из предоставленного указателя среды, во время выполнения программы. В CP/M такая среда не предусмотрена, поэтому, первый вызов **getenv()** попытается открыть файл в текущей области пользователя на текущем диске с названием ENVIRON. Этот файл должен содержать определения для любых переменных среды, которые должны быть доступны программе, например.

HITECH=0:C:

Определение каждой переменной должно быть на отдельной строке, состоящей из имени переменной (обычно все в верхнем регистре), затем без пробела следует знак равенства ('=') после чего значение, которое будет присвоено этой переменной.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

void main (void) {

printf("comspec = %s\n", getenv("COMSPEC"));

}

***Возвращаемое значение:***

NULL, если указанная переменная не может быть найдена.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**GETS**

***Синтаксис:***

**#include <stdio.h>**

**char \* gets (char \* s)**

***Описание:***

Функция **gets()** читает строку из стандартного ввода в буфер **s**, удаляя символ новой строки (см. описание fgets()). Буфер оканчивается нулевым символом. Во встроенной системе, **gets()** эквивалентно cgets() и getche() вызывается несколько раз для получения символов. Доступно редактирование (с помощью клавиши Backspace).

***Пример:***

#include <stdio.h>

void main (void) {

char buf[80];

printf("Type a line: ");

if(gets(buf))

puts(buf);

}

***См. также***

[fgetc()](#f_fgetc), [freopen()](#f_freopen), [puts()](#f_puts)

***Возвращаемое значение:***

Возвращает аргумент или значение NULL в конце файла.

**GETUID**

***Синтаксис:***

**#include <sys.h>**

**int getuid (void)**

***Описание:***

Функция **getuid()** возвращает текущий номер пользователя. В CP/M текущий номер пользователя определяет номер пользователя, связанный с открытым или созданным файлом, если только он не переопределяется явным префиксом номера пользователя в имени файла.

***Пример:***

#include <stdio.h>

#include <sys.h>

void main (void) {

printf("Current user number is: %d\n", getuid());

}

***См. также***

[setuid()](#f_setuid), [open()](#f_open)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**GETW**

***Синтаксис:***

**#include <stdio.h>**

**int getw (FILE \* stream)**

***Описание:***

Функция **getw()** возвращает одно слово (16 бит для Z80 и 8086) из назначенного потока **stream**. В конце файла возвращается EOF, но поскольку это нормальное слово, для тестирования EOF следует использовать макрос feof(). При чтении слова не требуется специального выравнивания в файле, так как чтение выполняется двумя последовательными вызовами getc(). Однако порядок байтов не определен. Читаемое слово должно быть записано putw(). Не полагайтесь на эту функцию для чтения двоичных данных, записанных другой программой.

***Пример:***

#include <stdio.h>

#include <sys.h>

void main (void) {

FILE \* fp;

int c;

fp = fopen("file.dat", "r");

if(!feof(fp)) {

while((c = getw(fp)) != EOF)

putw(c, stdout);

fclose(fp);

}

}

***См. также***

[putw()](#f_putw), [getc()](#f_getc), [fopen()](#f_fopen), [fclose()](#f_fclose)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**GMTIME**

***Синтаксис:***

**#include <time.h>**

**struct tm \* gmtime (time\_t \* t)**

***Описание:***

Эта функция преобразует время, на которое указывает **t**, которое выражается в секундах с 00:00:00 на 1 января 1970, в разобранное время, хранящееся в структуре, как определено в **time.h**. Структура определена в секции "Типы данных" (ниже).

***Пример:***

#include <stdio.h>

#include <time.h>

void main (void) {

time\_t clock;

struct tm \* tp;

time(&clock);

tp = gmtime(&clock);

printf("It’s %d in London\n", tp->tm\_year+1900);

}

***См. также***

[ctime()](#f_ctime), [asctime()](#f_astime), [time()](#f_time), [localtime()](#f_localtime)

***Возвращаемое значение:***

Возвращает структуру типа **tm**.

***Примечание:***

В примере требуется, чтобы пользователь предоставил процедуру time(), поскольку она не может быть предоставлена компилятором. См. описание [time()](#f_time) для получения дополнительной информации.

***Типы данных:***

typedef long time\_t;

struct tm {

int tm\_sec;

int tm\_min;

int tm\_hour;

int tm\_mday;

int tm\_mon;

int tm\_year;

int tm\_wday;

int tm\_yday;

int tm\_isdst;

};

**IM**

***Синтаксис:***

**#include <intrpt.h>**

**void im (unsigned char mode)**

***Описание:***

Эта функция позволяет установить режим прерывания процессора Z80. Аргументом **im()** является номер режима: 0, 1 или 2. Если для режима установлено значение 2, то в качестве побочного эффекта эта процедура устанавливает регистр **I** для указания начала таблицы векторов прерываний.

***Пример:***

#include <intrpt.h>

void main (void) {

im(2);

.

.

.

}

***См. также***

[ROM\_VECTOR()](#f_ROM_VECTOR), [RAM\_VECTOR()](#f_RAM_VECTOR), [CHANGE\_VECTOR()](#f_CHANGE_VECTOR), [set\_vector()](#f_set_vector)

**ISALNUM, ISALPHA, ISDIGIT, ISLOWER и другие**

***Синтаксис:***

**#include <ctype.h>**

**isalnum(char c)**

**isalpha(char c)**

**isascii(char c)**

**iscntrl(char c)**

**isdigit(char c)**

**islower(char c)**

**isprint(char c)**

**isgraph(char c)**

**ispunct(char c)**

**isspace(char c)**

**isupper(char c)**

***Описание:***

Эти макросы, определенные в файле **ctype.h**, проверяют предоставленный символ на принадлежность к одной из нескольких перекрывающихся групп символов. Обратите внимание, что все макросы кроме **isascii()** определены для **c**, если **isascii(c)** является истиной или если **c=EOF**.

**isalnum(c)** c является алфавитно-цифровым

**isalpha(c)** c является A-Z или a-z

**isascii(c)** c является 7-битным символом ASCII

**iscntrl(c)** c является управляющим символом

**isdigit(c)** c является десятичной цифрой

**islower(c)** c является a-z

**isprint(c)** c является печатаемым символом

**isgraph(c)** c является печатаемым символом, но не пробелом

**ispunct(c)** c является знаком пунктуации

**isspace(c)** c является пробелом, табуляцией или новой строкой

**isupper(c)** c является A-Z

**isxdigit(c)** c является 0-9 или a-f или A-F

***Пример:***

#include <ctype.h>

#include <stdio.h>

void main (void) {

char buf[80];

int i;

gets(buf);

i = 0;

while(isalnum(buf[i]))

i++;

buf[i] = 0;

printf("'%s' is the word\n", buf);

}

***См. также***

[toupper()](#f_toupper), [tolower()](#f_tolower), [toascii()](#f_toascii)

**ISATTY**

***Синтаксис:***

**#include <unixio.h>**

**int isatty (int fd)**

***Описание:***

Тестирует тип файла, связанного с **fd**. Она возвращает true, если файл присоединен к терминальному устройству. Обычно она используется для тестирования осуществления стандартного ввода из файла или консоли. Для тестирования потоков stdio используйте **isatty(fileno(stream)**).

***Пример:***

#include <unixio.h>

#include <stdio.h>

void main (void) {

if(isatty(fileno(stdin)))

printf("input not redirected\n");

else

printf("Input is redirected\n");

}

***Возвращаемое значение:***

Ноль, если поток связан с файлом. Единица, если он связан с консолью или другим устройством типа клавиатуры.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**KBHIT**

***Синтаксис:***

**#include <conio.h>**

**int kbhit (void)**

***Описание:***

Эта функция возвращает 1, если символ был нажат на клавиатуре консоли, 0 в противном случае. Обычно символ затем считывается с помощью getch().

***Пример:***

#include <conio.h>

void main (void) {

int i;

while(!kbhit()) {

cputs("I'm waiting..");

for(i = 0 ; i != 1000 ; i++)

continue;

}

}

***См. также***

[getch()](#f_getch), [getche()](#f_getche)

***Возвращаемое значение:***

Возвращается единица, если символ был нажат на консольной клавиатуре, иначе ноль.

**LDEXP**

***Синтаксис:***

**#include <math.h>**

**double ldexp (double f, int i)**

***Описание:***

Функция **ldexp()** выполняет обратную операцию frexp(). Целое число **i** добавляется к экспоненте числа с плавающей точкой **f** и результат возвращается.

***Пример:***

#include <math.h>

#include <stdio.h>

void main (void) {

double f;

f = ldexp(1.0, 10);

printf("1.0 \* 2^10 = %f\n", f);

}

***См. также***

[frexp()](#f_frexp)

***Возвращаемое значение:***

Возвращаемое значение - значения с плавающей точкой **f** к экспоненте которого добавлено целое число **i.**

**LDIV**

***Синтаксис:***

**#include <stdlib.h>**

**ldiv\_t ldiv (long number, long denom)**

***Описание:***

Подпрограмма **ldiv()** делит числитель на знаменатель, вычисляя частное и остаток. Знак частного совпадает со знаком математического частного. Его абсолютное значение - самое большое целое число, которое является меньше, чем абсолютное значение математического частного.

Функция **ldiv()** подобна функции div(), различие состоит в том, что аргументы и члены возвращаемой структуры имеют тип long int.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

void main (void) {

ldiv\_t lt;

lt = ldiv(1234567, 12345);

printf("Quotient = %ld, remainder = %ld\n", lt.quot, lt.rem);

}

***См. также***

[div()](#f_div)

***Возвращаемое значение:***

Возвращает структуру типа **ldiv\_t.**

***Типы данных:***

typedef struct {

long quot; /\* частное \*/

long rem; /\* остаток \*/

} ldiv\_t;

**LOCALTIME**

***Синтаксис:***

**#include <time.h>**

**struct tm \* localtime (time\_t \* t)**

***Описание:***

Функция **localtime()** преобразует время, на которое указывает **t**, выраженное в секундах с 00:00:00 1 января 1970 года, в разобранное время, хранящееся в структуре, как определено в **time.h**. Процедура **localtime()** учитывает содержимое глобального целого числа time\_zone. Оно должно содержать количество минут, в течение которых локальный часовой пояс находится на западе от Гринвича. Поскольку в MS-DOS нет способа фактически предопределить это значение, **localtime()** по умолчанию возвращает тот же результат, что и gmtime().

***Пример:***

#include <stdio.h>

#include <time.h>

char \* wday[] = {

"Sunday", "Monday", "Tuesday", "Wednesday",

"Thursday", "Friday", "Saturday"

void main (void) {

time\_t clock;

struct tm \* tp;

time(&clock);

tp = localtime(&clock);

printf("Today is %s\n", wday[tp->tm\_wday]);

}

***См. также***

[ctime()](#f_ctime), [asctime()](#f_astime), [time()](#f_time)

***Возвращаемое значение:***

Возвращает структуру типа **tm.**

***Примечание:***

В примере требуется, чтобы пользователь предоставил процедуру time(), поскольку она не может быть предоставлена компилятором. См. описание [time()](#f_time) для получения дополнительной информации.

***Типы данных:***

typedef long time\_t;

struct tm {

int tm\_sec;

int tm\_min;

int tm\_hour;

int tm\_mday;

int tm\_mon;

int tm\_year;

int tm\_wday;

int tm\_yday;

int tm\_isdst;

};

**LOG, LOG10**

***Синтаксис:***

**#include <math.h>**

**double log (double f)**

**double log10 (double f)**

***Описание:***

Функция **log()** возвращает натуральный логарифм **f**. Функция **log10()** возвращает десятичный логарифм **f**.

***Пример:***

#include <math.h>

#include <stdio.h>

void main (void) {

double f;

for(f = 1.0 ; f <= 10.0 ; f += 1.0)

printf("log(%1.0f) = %f\n", f, log(f));

}

***См. также***

[exp()](#f_exp), [pow()](#f_pow)

***Возвращаемое значение:***

Нуль, если параметр отрицателен.

**LONGJMP**

***Синтаксис:***

**#include <setjmp.h>**

**void longjmp (jmp\_buf buf, int val)**

***Описание:***

Функция **longjmp()**, в сочетании с setjmp(), обеспечивает механизм для не-локальных переходов. Чтобы использовать это средство, нужно вызвать setjmp() с параметром **jmp\_buf** на уровне некоторой внешней функции. Вызов из setjmp() возвращает 0.

Чтобы вернуться к этому уровню выполнения, она может быть вызвана с тем же параметром **jmp\_buf** из внутреннего уровня выполнения. Однако следует отметить, что функция, которая вызывается setjmp() должна быть активной во время вызова **longjmp()**. Нарушение этого правила вызовет аварийную ситуацию, из-за использования стека, содержащего недопустимые данные. Параметр **val** **longjmp()** будет значением, очевидно возвращенным из setjmp(). Оно должно обычно быть ненулевым, чтобы отличить его от подлинного вызова setjmp().

***Пример:***

#include <stdio.h>

#include <setjmp.h>

#include <stdlib.h>

void inner (void) {

longjmp(jb, 5);

}

void main (void) {

int i;

if(i = setjmp(jb)) {

printf("setjmp returned %d\n", i);

exit(0);

}

printf("setjmp returned 0 - good\n");

printf("calling inner...\n");

inner();

printf("inner returned - bad!\n");

}

***См. также***

[setjmp()](#f_setjmp)

***Возвращаемое значение:***

Подпрограмма **longjmp()** никогда не возвращается.

***Примечание:***

Функция, вызываемая setjmp() должна оставаться активной при вызове **longjmp()**. Нарушение этого правила вызовет катастрофу из-за использования стека, содержащего недопустимые данные.

**LSEEK**

***Синтаксис:***

**#include <unixio.h>**

**long lseek (int fd, long offs, int wh)**

***Описание:***

Эта функция работает аналогично fseek(), однако она делает это на не буферизированных файловых дескрипторах ввода-вывода низкого уровня, а не на потоках stdio. Он также возвращает результирующее местоположение указателя. Таким образом, **lseek(fd, 0L, 1)** возвращает текущее местоположение указателя, не перемещая его.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

#include <unixio.h>

void main (void) {

int fd;

fd = open("test.fil", 1); /\* открыть для записи \*/

if(fd < 0)

exit(1);

lseek(fd,0L, 2); /\* стремиться к концу \*/

write(fd, "more stuff\r\n", 12);

close(fd);

}

***См. также***

[open()](#f_open), [close()](#f_close), [read()](#f_read), [write()](#f_write)

***Возвращаемое значение:***

Минус единица (-1) возвращен при ошибке, в противном случае полученное расположение.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**MALLOC**

***Синтаксис:***

**#include <stdlib.h>**

**void \* malloc (size\_t cnt)**

***Описание:***

Функция **malloc()** пытается выделить **cnt** байтов памяти из "кучи", динамически распределяемой области памяти. Если вызов успешный, она возвращает указатель на выделенный блок, иначе возвращается 0. Выделенная память может быть освобождена с помощью free() или изменена в размере используя realloc(). Функция **malloc()** вызывает sbrk(), чтобы получить память, и она в свою очередь вызывает calloc(). Функция **malloc()** не очищает память, которую она получает, в отличие от calloc().

***Пример:***

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

void main (void) {

char \* cp;

cp = malloc(80);

if(!cp)

printf("Malloc failed\n");

else {

strcpy(cp, "a string");

printf("block = '%s'\n", cp);

free(cp);

}

}

***См. также***

[calloc()](#f_calloc), [free()](#f_free), [realloc()](#f_reallok)

***Возвращаемое значение:***

Указатель на память, если она выполнена успешно. В противном случае - NULL.

**MEMCHR**

***Синтаксис:***

**#include <string.h>**

**void \* memchr (const void \* block, int val, size\_t length)**

***Описание:***

Функция **memchr()** аналогична функции strchr(), за исключением того, что вместо поиска в строке завершенной нулем, она ищет указанный байт в блоке памяти определенной длины. Ее аргументы представляют собой указатель на память в которой будет осуществляться поиск, значение искомого байта и длину блока. Возвращается указатель на первое вхождение этого байта в блок.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

unsigned int ary[] = {1, 5, 0x6789, 0x23};

void main (void) {

char \* cp;

cp = memchr(ary, 0x89, sizeof ary);

if(!cp)

printf("not found\n");

else

printf("Found at offset %u\n", cp - (char \*)ary);

}

***См. также***

[strchr()](#f_strchr)

***Возвращаемое значение:***

Указатель на первый байт, соответствующий аргументу, если он существует. В противном случае - NULL.

**MEMCMP**

***Синтаксис:***

**#include <string.h>**

**int memcmp (const void \* s1, const void \* s2, size\_t n)**

***Описание:***

Функция **memcmp()** сравнивает два блока памяти, длиной **n**, и возвращает значение со знаком, подобное strncmp(). В отличие от strncmp() сравнение не останавливается на нулевом символе. Для сравнения используется упорядоченная последовательность символов ASCII, но эффект включения символов, не являющихся ASCII, в блоки памяти в смысле возвращаемого значения не определен. Проверка на равенство всегда надежна.

***Пример:***

#include <stdio.h>

#include <string.h>

unsigned int ary[] = {1, 5, 0x6789, 0x23};

void main (void) {

int buf[10], cow[10], i;

buf[0] = 1; cow[0] = 1;

buf[1] = 3; cow[1] = 3;

buf[2] = 4; cow[2] = 5;

i = memcmp(buf, cow, 3\*sizeof(int));

if(i < 0)

printf("Less than\n");

else if(i > 0)

printf("Greater than\n");

else

printf("Equal\n");

}

***См. также***

[strncpy()](#f_strncpy), [strncmp()](#f_strncmp), [strchr()](#f_strchr), [memset()](#f_memset), [memchr()](#f_memchr)

***Возвращаемое значение:***

Возвращает минус единицу, нуль или единицу, в зависимости от того, указывает ли **s1** на строку, которая меньше, равна или больше, чем строка, на которую указывает **s2** в последовательности сортировки.

**MEMCPY**

***Синтаксис:***

**#include <string.h>**

**void \* memcpy (void \* d, const void \* s, size\_t n)**

***Описание:***

Функция **memcpy()** копирует **n** байтов памяти, начиная с местоположения, на которое указывает **s**, в блок памяти, на который указывает **d**. Результат копирования перекрывающихся блоков не определен. Функция **memcpy()** отличается от strcpy(), тем что копирует заданное количество байтов, а не все байты до нулевого разделителя.

***Пример:***

#include <string.h>

#include <stdio.h>

void main (void) {

char buf[80];

memset(buf, 0, sizeof buf);

memcpy(buf, "a partial string", 10);

printf("buf = '%s'\n", buf);

}

***См. также***

[strncpy()](#f_strncpy), [strncmp()](#f_strncmp), [strchr()](#f_strchr), [memset()](#f_memset)

***Возвращаемое значение:***

Процедура **memcpy()** возвращает свой первый аргумент.

**MEMMOVE**

***Синтаксис:***

**#include <string.h>**

**void \* memmove (void \* s1, const void \* s2, size\_t n)**

***Описание:***

Функция **memmove()** подобна функции memcpy() за исключением того, что копирование перекрывающихся блоков обрабатывается правильно. Т.е. она копирует вперед или назад, если это необходимо, чтобы правильно скопировать один блок в другой, который перекрывает его.

***См. также***

[strncpy()](#f_strncpy), [strncmp()](#f_strncmp), [strchr()](#f_strchr), [memcpy()](#f_memcpu)

***Возвращаемое значение:***

Функция **memmove()** возвращает свой первый аргумент.

**MEMSET**

***Синтаксис:***

**#include <string.h>**

**void \* memset (void \* s, int c, size\_t n)**

***Описание:***

Функция **memset()** заполняет **n** байтов памяти, начинающихся в расположении, на которое указывает **s** байтом **c**.

***Пример:***

#include <string.h>

#include <stdio.h>

void main (void) {

char abuf[20];

strcpy(abuf, "This is a string");

memset(abuf, 'x', 5);

printf("buf = '%s'\n", abuf);

}

***См. также***

[strncpy()](#f_strncpy), [strncmp()](#f_strncmp), [strchr()](#f_strchr), [memcpy()](#f_memcpu), [memchr()](#f_memchr)

**MODF**

***Синтаксис:***

**#include <math.h>**

**double modf (double value, double \* iptr)**

***Описание:***

Функция **modf()** разделяет значение аргумента на целочисленную и дробную части, каждая из которых имеет тот же знак, что и значение. Например, -3.17 будет разбита на целочисленную часть (-3) и дробную часть (-0,17).

Целочисленная часть хранится как двойной объект типа double, на который указывает **iptr**.

***Пример:***

#include <math.h>

#include <stdio.h>

void main (void) {

double i\_val, f\_val;

f\_val = modf( -3.17, &i\_val);

}

***Возвращаемое значение:***

Дробная часть со знаком значения **value**.

**OPEN**

***Синтаксис:***

**#include <unixio.h>**

**int open (const char \* name, int mode)**

***Описание:***

Функция **open()** является основным средством открытия файлов для чтения и записи. Осуществляется поиск файла с указанным именем **name**, и если найден, он открывается для чтения, записи или того и другого. Режим **mode** кодируется следующим образом:

**Режим Значение**

**0** Открытие только для чтения

**1** Открытие только для записи

**2** Открытие для чтения и записи

Файл должен уже существовать - если это не так, должен использоваться вызов creat() для его создания. При успешном открытии возвращается дескриптор файла. Он является неотрицательным целым числом, которое может использоваться впоследствии для обращения к открытому файлу. Если открыть файл не удалось, возвращается -1. В MS-DOS синтаксис имен файлов является стандартным MS-DOS. Синтаксис имени файла CP/M:

**[uid:][drive:]name.type**

где **uid** десятичное число от 0 до 15, **drive** - буква диска от A до **P** или от **a** до **p**, **name** - имя из 1-8 символов и **type** - расширение от 0 до 3 символов. Хотя существует несколько ограничений для символов, относящихся к имени и расширению файла, рекомендуется, чтобы они были ограничены буквенно-цифровыми и стандартными печатаемым символами. Использование странных символов может вызвать проблемы при доступе и/или удалении файла.

Один или оба **uid:** и **drive:** могут быть опущены. Если предоставлены оба, **uid:** должен быть на первом месте. Обратите внимание, что [ и ] являются только мета-символами. Некоторые примеры:

**fred.dat**

**file.c**

**0:xyz.com**

**0:a:file1.p**

**a:file2.**

Если опущен **uid:**, файл будет разыскиваться с uid, равным текущему номеру пользователя, которое возвращает getuid(). Если опущен **drive:**, файл будет разыскиваться на выбранном в настоящее время диске. Распознаются следующие специальные имена файлов:

**lst:** Получает доступ к устройству печати - только для записи

**pun:** Получает доступ к устройству перфорации - только для записи

**rdr:** Получает доступ к считывающему устройству - только для чтения

**con:** Получает доступ к системной консоли - чтение-запись

Имена файлов могут быть в любом регистре - они преобразуются в верхний регистр во время обработки имени.

Имена файлов MS-DOS могут быть любым допустимыми в MS-DOS 2xx именами файлов, например:

**fred.nrk**

**A:\HITECH\STDIO.H**

Специальные имена устройств (например, CON, LST) также распознаются. Они не требуют (и не должны иметь) завершающего двоеточия.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

#include <unixio.h>

void main (void) {

int fd;

fd = open("test.fil", 1); /\* открыть для записи \*/

if(fd < 0)

exit(1);

lseek(fd,0L, 2); /\* стремиться к концу \*/

write(fd, "more stuff\r\n", 12);

close(fd);

}

***См. также***

[close()](#f_close), [fopen()](#f_fopen), [fclose()](#f_fclose), [read()](#f_read), [write()](#f_write), [creat()](#f_creat)

***Возвращаемое значение:***

В случае успеха - положительное число (файловый дескриптор) или минус единицу (-1), если она терпит неудачу.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**PERROR**

***Синтаксис:***

**#include <stdio.h>**

**void perror (const char \* s)**

***Описание:***

Эта процедура распечатает в поток stderr аргумент **s**, с последующим сообщением с подробным описанием последней ошибки, возвращаемой системным вызовом DOS. Номер ошибки извлекается из глобальной переменной **errno**. Подпрограмма **perror()** мало полезна в CP/M, поскольку она не дает столько информации об ошибках, сколько в MS-DOS.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

void main (int argc, char \*\* argv) {

if(argc < 2)

exit(0);

if(!freopen(argv[1], "r", stdin))

perror(argv[1]);

}

***См. также***

strerror()

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**PERSIST\_CHECK, PERSIST\_VALIDATE**

***Синтаксис:***

**#include <sys.h>**

**int persist\_check (int flag)**

**void persist\_validate (void)**

***Описание:***

Функция **persist\_check()** используется с энергонезависимым ОЗУ, объявленным с помощью квалификатора persistent. Она проверяет область NVRAM, используя магическое число, сохраненное в скрытой переменной предыдущим вызовом **persist\_validate()** и контрольную сумму, также вычисляемую функцией **persist\_validate()**. Если магическое число и контрольная сумма верны, она возвращает true (отличное от нуля). Если они неверны, она возвращает ноль. В этом случае она дополнительно обнулит и повторно проверит энергонезависимую область ОЗУ (вызывая **persist\_validate()**). Это выполняется, если аргумент **flag** имеет значение true.

Процедуру **persist\_validate()** следует вызывать после каждого изменения переменной, объявленной persistent. Она установит магическое число и пересчитает контрольную сумму.

***Пример:***

#include <sys.h>

#include <stdio.h>

persistent long reset\_count;

void main (void) {

if(!persist\_check(1))

printf("Reset count invalid - zeroed\n");

else

printf("Reset number %ld\n", reset\_count);

reset\_count++; /\* обновление счетчика \*/

persist\_validate(); /\* и контрольной суммы \*/

for(;;)

continue; /\* спать до следующего сброса \*/

}

***Возвращаемое значение:***

Значение false (ноль), если область энергонезависимого ОЗУ (NVRAM) недопустима; Значение true (не равно нулю), если область NVRAM является допустимой.

**POW**

***Синтаксис:***

**#include <math.h>**

**double pow (double f, double p)**

***Описание:***

Функция **pow()** возводит свой первый аргумент **f** в степень **p**.

***Пример:***

#include <math.h>

#include <stdio.h>

void main (void) {

double f;

for(f = 1.0 ; f <= 10.0 ; f += 1.0)

printf("pow(2, %1.0f) = %f\n", f, pow(2, f));

}

***См. также***

[log()](#f_log), [log10()](#f_log10), [exp()](#f_exp)

***Возвращаемое значение:***

Значение **f** в степень **p**.

**PRINTF, VPRINTF**

***Синтаксис:***

**#include <stdio.h>**

**int printf (const char \* fmt, ...)**

**#include <stdio.h>**

**#include <stdarg.h>**

**int vprintf (const char \* fmt, va\_list va\_arg)**

***Описание:***

**printf()** является программой форматированного вывода, воздействующей на stdout. Существуют аналогичные подпрограммы, воздействующие на заданный поток (fprintf()) или буфер строки (sprintf()). Подпрограмме **printf()** передается строка формата, за которым следует список из нуля или более параметров.

Каждая спецификация преобразования имеет форму **%m.nc**, где символ процента % представляет преобразование, за которым следует необязательная спецификация ширины **m**. **n** является дополнительной спецификацией точности (предваряется точкой), и c является буквой, определяющей тип преобразования.

Знак "минус" ('-') предшествующий **m** указывает выравнивание преобразуемого значения в поле по левому краю, а не по правому. Когда ширина поля больше, чем требуется для преобразования, дополнение пробелами выполняется слева или справа, согласно определения. Если определено числовое преобразование с выравниванием вправо и первая цифра **m** равна 0, то дополнение будет выполняться с помощью нулей, а не пробелов. Для целочисленных форматов точность указывает минимальное количество выводимых цифр, при этом, при необходимости, при формировании числа вставляются ведущие нули.

Если вместо десятичной константы используется символ \*, например, в формате %\*d, будет взят один целочисленный параметр из списка, чтобы обеспечить это значение. Типы преобразования:

**f** Плавающая точка - **m** является общей шириной, и **n** - число цифр после десятичной точки. Если **n** опущено его значение по умолчанию равняется 6. Если точность равна нулю, десятичная точка будет опущена, если не будет определен альтернативный формат.

**e** Печатать соответствующего параметра в экспоненциальном представлении. В остальном аналогичен **f**.

g Используется формат e или f, в зависимости от того, какая максимальная точность дает минимальную ширину. Любые завершающие нули после десятичной точки будут удалены, и если после десятичной точки не останется цифр, они также будут удалены.

**oxXud** Целочисленное преобразование в системе счисления 8, 16, 16, 10 и 10 соответственно. Преобразование со знаком в случае **d**, без знака в противном случае. Значение точности определяет общее количество распечатываемых цифр и может использоваться, чтобы вызвать печать предшествующих нулей. Например, %8.4x распечатает, по крайней мере, 4 шестнадцатеричных цифры в поле шириной 8 символов. Предшествование ключевой буквы **l** указывает, что параметр значения является длинным целым или значением без знака. Буква X распечатывает шестнадцатеричные числа, используя прописные буквы A-F, а не a-f, как было бы распечатано при использовании x. Если задан альтернативный формат, то для восьмеричного формата будет задано начальное нулевое значение, а для шестнадцатеричного формата - начальное 0x или 0X.

**s** печатать строки - значением параметра, как предполагается, является символьный указатель. В поле шириной из **m** символов будет распечатано не более **n** символов строки.

**c** Параметр, как предполагается, является единственным символом и распечатывается без преобразований.

Любые другие символы, используемые в качестве спецификации преобразования, будут напечатаны. Таким образом, %% будет производить одинарный знак процента.

Функция **vprintf()** подобна **printf()**, но принимает указатель на переменный список аргументов, а не список аргументов. См. описание [va\_start()](#f_va_start) для получения дополнительной информации о переменных списках аргументов. Пример использования **vprintf()** приведен ниже.

***Пример:***

printf("Total = %4d%%", 23)

создаст 'Total = 23%'

printf("Size is %lx" , size)

Когда size имеет тип данных long, выводит size в шестнадцатеричном представлении.

printf("Name = %.8s", "a1234567890")

создаст 'Name = a1234567'

printf("xx%\*d", 3, 4)

создаст 'xx 4'

/\* Пример  vprintf \*/

#include <stdio.h>

int error (char \* s, ...) {

va\_list ap;

va\_start(ap, s);

printf("Error: ");

vprintf(s, ap);

putchar('\n');

va\_end(ap);

}

void main (void) {

int i;

i = 3;

error("testing 1 2 %d", i);

}

***См. также***

[fprintf()](#f_fprintf), [sprintf()](#f_sprintf)

***Возвращаемое значение:***

Функции **printf()** и **vprintf()** возвращают количество символов, записанных в **stdout**.

**PUTC**

***Синтаксис:***

**#include <stdio.h>**

**int putc (int c, FILE \* stream)**

***Описание:***

Функция **putc()** представляет собой макро версию fputc() и определяется в файле **stdio.h**. Он помещает предоставленный символ **c** в указанный поток ввода-вывода **stream**.

***Пример:***

#include <stdio.h>

char \* x = "this is a string";

void main (void) {

char \* cp;

cp = x;

while(\*x)

putc(\*x++, stdout);

putc('\n', stdout);

}

***См. также***

[fputc()](#f_fputc), [getc()](#f_fgetc), [fopen()](#f_fopen), [fclose()](#f_fclose), [putch()](#f_putch)

***Возвращаемое значение:***

Символ, передаваемый в качестве аргумента, или EOF при ошибке.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**PUTCH**

***Синтаксис:***

**#include <conio.h>**

**void putch (char c)**

***Описание:***

Функция **putch()** выводит символ **c** на экран консоли, добавляя возврат каретки, если символ является символом новой строки. В системе CP/M или MS-DOS она использует один из системных вызовов ввода-вывода. Во встроенной системе эта и другие, связанные с ней подпрограммы, зависят от оборудования. Стандартные процедуры **putch()** в библиотеке для встроенных систем имеют интерфейс с последовательным портом или отладчиком Lucifer.

***Пример:***

#include <conio.h>

char \* x = "this is a string";

void main (void) {

char \* cp;

cp = x;

while(\*x)

putch(\*x++);

putch('\n');

}

***См. также***

[cgets()](#f_cgets), [cputs()](#f_cputs), [getch()](#f_getch), [getche()](#f_getche)

**PUTCHAR**

***Синтаксис:***

**#include <stdio.h>**

**int putchar (int c)**

***Описание:***

Функция **putchar()** - это операция putc() на stdout, определенная в **stdio.h**.

***Пример:***

#include <stdio.h>

char \* x = "this is a string";

void main (void) {

char \* cp;

cp = x;

while(\*x)

putchar(\*x++);

putchar('\n');

}

***См. также***

[putc()](#f_putc), [getc()](#f_getc), [freopen()](#f_freopen), [fclose()](#f_fclose)

***Возвращаемое значение:***

Символ, передаваемый в качестве аргумента, или EOF при ошибке.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**PUTS**

***Синтаксис:***

**#include <stdio.h>**

**int puts (const char \* s)**

***Описание:***

Функция **puts()** записывает строку **s** в поток stdout, добавляя новую строку. Нулевой символ, заканчивающий строку, не копируется.

***Пример:***

#include <stdio.h>

void main (void) {

puts("Hello, world!");

}

***См. также***

[fputs()](#f_fputs), [gets()](#f_gets), [freopen()](#f_freopen), [fclose()](#f_fclose)

***Возвращаемое значение:***

При ошибке возвращается EOF, ноль в противном случае.

**PUTW**

***Синтаксис:***

**#include <stdio.h>**

**int putw (int w, FILE \* stream)**

***Описание:***

Функция **putw()** копирует слово w в указанный поток **stream**. Она возвращает **w**, за исключением ошибки, в этом случае возвращается EOF. Поскольку это хорошее целое число, для проверки ошибок следует использовать ferror(). Подпрограмма getw() может использоваться для чтения целых числах, записанных **putw()**.

***Пример:***

#include <stdio.h>

void main (void) {

FILE \* fp;

int c;

fp = fopen("file.dat", "r");

if(!feof(fp)) {

while((c = getw(fp)) != EOF)

putw(c, stdout);

fclose(fp);

}

}

***См. также***

[getw()](#f_getw), [fopen()](#f_fopen), [fclose()](#f_fclose)

***Возвращаемое значение:***

Возвращаемое значение - **w**, за исключением возникновения ошибки, тогда возвращаемое значение - EOF.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**QSORT**

***Синтаксис:***

**#include <stdlib.h>**

**void qsort (void \* base, size\_t nel, size\_t width,**

**int (\*func)(const void \*, const void \*))**

***Описание:***

Функция **qsort()** является реализацией алгоритма быстрой сортировки. Она сортирует массив из **nel** элементов, каждый длиной **width** байт, расположенных в памяти последовательно начиная с адреса **base**. Аргумент **func** является указателем на функцию, используемую функцией **qsort()** для сравнения элементов. Она вызывает func с указателями на два элемента для сравнения. Если первый элемент считается больше, равен или меньше второго, тогда func() должна возвратить значение больше 0, равно 0 или меньше 0 соответственно.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

int aray[] = {

567, 23, 456, 1024, 17, 567, 66

};

int sortem (const void \* p1, const void \* p2) {

return \*(int \*)p1 - \*(int \*)p2;

}

void main (void) {

register int i;

qsort(aray, sizeof aray/sizeof aray[0], sizeof aray[0], sortem);

for(i = 0 ; i != sizeof aray/sizeof aray[0] ; i++)

printf("%d\t", aray[i]);

putchar('\n');

}

***Примечание:***

Параметр функции должен быть указателем на объект типа функция, аналогичной: int func (const void \*, const void \*), то есть она должна принимать два параметра const void \* и должна быть объявлена.

**RAM\_VECTOR, CHANGE\_VECTOR, READ\_RAM\_VECTOR**

***Синтаксис:***

**#include <intrpt.h>**

**void RAM\_VECTOR (unsigned vector, isr func)**

**void CHANGE\_VECTOR (unsigned vector, isr func)**

**void (\* READ\_RAM\_VECTOR (unsigned vector)(void))**

***Описание:***

Макросы **RAM\_VECTOR()**, **CHANGE\_VECTOR()** и **READ\_RAM\_VECTOR()** используются для инициализации, изменения и чтения векторов прерываний, направленных через внутренние векторы прерываний на основе ОЗУ. Эти макросы должны использоваться только для векторов, которые необходимо модифицировать, чтобы указывать на разные функции прерывания в разных точках программы. Макросы **CHANGE\_VECTOR()** и **READ\_RAM\_VECTOR()** должны использоваться только с векторами прерываний, которые были инициализированы с использованием **RAM\_VECTOR()**, иначе будет возвращен мусор.

Дополнительную информацию см. в Разделе "[Обработка прерываний в C](#_Обработка_прерываний_в)" в этом руководстве.

***Пример:***

volatile unsigned char wait\_flag;

interrupt void wait\_handler(void) {

++wait\_flag;

}

void wait\_for\_serial\_intr(void) {

interrupt void (\*old\_handler)(void);

di();

old\_handler = READ\_RAM\_VECTOR(RXI);

wait\_flag = 0;

CHANGE\_VECTOR(RXI, wait\_handler);

}

***См. также***

[di()](#f_di), [ei()](#f_ei), [ROM\_VECTOR()](#f_ROM_VECTOR)

***Примечание:***

Эти макросы, для Z80/Z180, могут использоваться только с 2 режимом прерывания.

**RAND**

***Синтаксис:***

**#include <stdlib.h>**

**int rand (void)**

***Описание:***

Функция **rand()** является генератором псевдослучайных чисел. Она возвращает целое число в диапазоне от 0 до 32767, которое изменяется в псевдослучайном способом при каждом вызове. Алгоритм будет генерировать детерминированную последовательность, если начать с той же точки. Начальная точка устанавливается с помощью вызова srand(). В этом примере показано использование функции time() для генерации другой начальной точки для каждой последовательности.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

#include <time.h>

void main (void) {

time\_t toc;

int i;

time(&toc);

srand((int)toc);

for(i = 0 ; i != 10 ; i++)

printf("%d\t", rand());

putchar('\n');

}

***См. также***

[srand()](#f_srand)

***Примечание:***

В примере требуется, чтобы пользователь предоставил процедуру time(), поскольку она не может быть предоставлена компилятором. См. описание [time()](#f_time) для получения дополнительной информации.

**READ**

***Синтаксис:***

**#include <unixio.h>**

**size\_t read (int fd, void \* buf, size\_t cnt)**

***Описание:***

Функция **read()** читает из файла, связанного с **fd** до **cnt** байтов в буфер, рас-положенный в **buf**. Она возвращает число фактически прочитанных байтов. Возврат 0 указывает конец файла. Возврат отрицательного значения означает ошибку. Значение **fd** должно быть получено из предыдущего вызова open(). Функция **read()** может вернуть меньше байтов, чем требуется, например, при чтении из консоли, в этом случае **read()** считает одну входную строку.

***Пример:***

#include <stdio.h>

#include <unixio.h>

void main (void) {

char buf[80];

int i;

/\* чтение из stdin \*/

i = read(0, buf, sizeof(buf));

printf("Read %d bytes\n", i);

}

***См. также***

[open()](#f_open), [close()](#f_close), [write()](#f_write)

***Возвращаемое значение:***

Число прочитанных байт. Ноль в конце файла, минус единица (-1) при ошибке. Будьте осторожны, чтобы не интерпретировать прочитанное значение > 32767 как возвращаемое отрицательное значение.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**REALLOC**

***Синтаксис:***

**#include <stdlib.h>**

**void \* realloc (void \* ptr, size\_t cnt)**

***Описание:***

Функция **realloc()** освобождает блок памяти расположенный по адресу **ptr**, который должен был быть получен предыдущим вызовом malloc(), calloc() или realloc(), затем пытается выделить **cnt** байт динамической памяти, и если удачно, то скопирует содержимое блока памяти в располагающийся в **ptr** новый блок.

В лучшем случае **realloc()** скопирует количество байтов, которые были в старом блоке, но если новый блок будет меньше, то будут скопированы только **cnt** байтов.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

#include <unixio.h>

void main (void) {

char \* cp;

cp = malloc(255);

if(gets(cp))

cp = realloc(cp, strlen(cp)+1);

printf("buffer now %d bytes long\n", strlen(cp)+1);

}

***См. также***

[malloc()](#f_malloc), [calloc()](#f_calloc)

***Возвращаемое значение:***

Указатель на новый (или измененный) блок. Значение NULL, если блок не может бы быть расширен. Запрос на сжатие блока никогда не завершится ошибкой.

**REMOVE**

***Синтаксис:***

**#include <stdio.h>**

**int remove (const char \* s)**

***Описание:***

Функция **remove()** попытается удалить файл, названный параметром **s** из каталога.

***Пример:***

#include <stdio.h>

void main (void) {

if(remove("test.fil") < 0)

perror("test.fil");

}

***См. также***

[unlink()](#f_unlink)

***Возвращаемое значение:***

Ноль в случае успеха, минус единица (-1) при ошибке.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**RENAME**

***Синтаксис:***

**#include <stdio.h>**

**int rename (const char \* name1, const char \* name2)**

***Описание:***

Файл с именем **name1** будет переименован в **name2**.

***Пример:***

#include <stdio.h>

void main (void) {

if(rename("test.fil", "test1.fil"))

perror("Rename");

}

***См. также***

[open()](#f_open), [close()](#f_close), [unlink()](#f_unlink)

***Возвращаемое значение:***

Будет возвращена минус единица (-1), если переименование не было успешно, нуль, если переименование было выполнено.

***Примечание:***

Переименование не разрешено через диски или каталоги.

Эта процедура не может использоваться в системе на основе ПЗУ.

**REWIND**

***Синтаксис:***

**#include <stdio.h>**

**int rewind (FILE \* stream)**

***Описание:***

Эта функция попытается изменить местоположение указателя чтения-записи назначенного потока **stream** к началу файла. Этот вызов эквивалентен fseek(stream, 0L, 0).

***Пример:***

#include <stdio.h>

#include <stdlib.h>

void main (void) {

char buf[80];

if(!freopen("test.fil", "r", stdin))

exit(1);

gets(buf);

printf("got ’%s’\n", buf);

rewind(stdin);

gets(buf);

printf("Got '%s' again\n", buf);

}

***См. также***

[fseek()](#f_fseek), [ftell()](#f_ftell)

***Возвращаемое значение:***

Возвращаемое значение минус единица (-1) указывает, что попытка не была успешна, возможно потому, что поток связан не с файлом произвольного доступа, таким как символьное устройство.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**ROM\_VECTOR**

***Синтаксис:***

**#include <intrpt.h>**

**void ROM\_VECTOR (unsigned vector, isr func)**

***Описание:***

Макрос **ROM\_VECTOR()** используется, чтобы установить "твердый кодированный" вектор ROM, который указывает на обработчики прерываний. Этот макрос не генерирует кода, который исполняется во времени выполнения, поэтому его можно поместить в любом месте кода. Макрос **ROM\_VECTOR()** генерирует встроенный ассемблерный код, поэтому передаваемый ему адрес вектора может быть в любом формате, приемлемом для ассемблера.

Дополнительную информацию см. в Разделе "[Обработка прерываний в C](#_Обработка_прерываний_в)" в этом руководстве.

***См. также***

[di()](#f_di), [ei()](#f_ei), [RAM\_VECTOR()](#f_RAM_VECTOR)

***Примечание:***

Эти макросы, для Z80/Z180, могут использоваться только с 2 режимом прерывания.

**SBRK**

***Синтаксис:***

**#include <sys.h>**

**void \* sbrk (int incr)**

***Описание:***

Функция **sbrk()** увеличивает текущий адрес самой верхней ячейки памяти, выделенной программе на **incr** байт. Она возвращает указатель на предыдущее значение адреса самой верхней ячейки памяти. Таким образом, **sbrk(0)** возвращает указатель на текущий адрес самой верхней ячейки памяти, не изменяя его значение. Это низкоуровневая процедура, не предназначенная для вызова пользовательским кодом. Вместо нее используйте malloc().

***См. также***

brk(), [malloc()](#f_malloc), [calloc()](#f_calloc), [realloc()](#f_reallok), [free()](#f_free)

***Возвращаемое значение:***

Если есть недостаточная память, чтобы удовлетворить запрос, возвращается минус единица (-1).

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**SCANF, VSCANF**

***Синтаксис:***

**#include <stdio.h>**

**int scanf (const char \* fmt, ...)**

**#include <stdio.h>**

**#include <stdarg.h>**

**int vscanf (const char \*, va\_list ap)**

***Описание:***

Функция **scanf()** выполняет форматированный ввод ("де-редактирование") из потока stdin. Аналогичные функции доступны для потоков и для строк. Функция **vscanf()** подобна, но принимает указатель на список аргументов, а не список дополнительных параметров. Этот указатель должен быть предварительно инициализирован с помощью va\_start().

Входные преобразования выполняются согласно строке **fmt**. Обычно символ в строке формата должен соответствовать символу на вводе. Однако, пробелу в строке формата будет соответствовать ноль или более "пробельных" символов на вводе, т.е. символов пробела, табуляции или новой строки.

Спецификация преобразования принимает форму символа **%**, за которым необязательно следует символ подавления присвоения ('**\***'), затем необязательное числовое значение максимальной ширины поля, за которым следует символ спецификации преобразования. Каждая спецификация преобразования, если она не включает символ подавления присвоения, присвоит значение переменной, на которую указывает следующий параметр. Таким образом, если присутствуют две спецификации преобразования в строке **fmt**, должно быть два дополнительных указателя параметров. Символы преобразования следующие:

**o x d** Пропустит пробельные символы, затем преобразует число к основанию системы счисления 8, 16 или 10 соответственно. Если была предоставлена ширина поля, возьмет символы из ввода в количестве не более этого числа. При вводе распознается предшествующий знак "минус".

**f** Пропустит пробельные символы, затем преобразует вещественное число в стандартном или в экспоненциальном представлении. Ширина поля применяется как указано выше.

**s** Пропустит пробельные символы, затем скопирует последовательность максимальной длины из непробельных символов. Параметр указателя должен быть указателем на char. Ширина поля будет ограничивать количество скопированных символов. Результирующая строка будет завершена 0.

**c** Копирует следующий символ из ввода. Параметр указателя, как пред-полагается, является указатель на char. Если ширина поля определена, то будут скопированы символы в количестве не более этого числа. Это отличается от формата **s** тем, что пробельные символы не прерывают последовательность символов.

Символам преобразования **o**, **x**, **u**, **d** и f может предшествовать **l**, чтобы указать, что соответствующий параметр указателя является указателем на long или double при необходимости. Предшествующий **h** означает, что параметр указателя является указателем на short, а не int.

***Пример:***

**scanf("%d %s", &a, &c)**

при вводе " 12s"

12 будет присвоено **a** и "s" - **s**.

**scanf("%3cd %lf", &c, &f)**

при вводе " abcd -3.5"

" abc" будет присвоено **c**, и -3.5 - **f**.

***См. также***

[fscanf()](#f_fscanf), [sscanf()](#f_sscanf), [printf()](#f_printf), [va\_arg()](#f_va_arg)

***Возвращаемое значение:***

Функция **scanf()** возвращает число успешных преобразований. Если до выполнения любых преобразований встречается конец файла, возвращается EOF.

**SETJMP**

***Синтаксис:***

**#include <setjmp.h>**

**int setjmp (jmp\_buf buf)**

***Описание:***

Функция **setjmp()** используется с longjmp() для нелокальных переходов. Для получения дополнительной информации см. описание [longjmp()](#f_longjmp).

***Пример:***

#include <stdio.h>

#include <setjmp.h>

#include <stdlib.h>

jmp\_buf jb;

void inner (void) {

longjmp(jb, 5);

}

void main (void) {

int i;

if(i = setjmp(jb)) {

printf("setjmp returned %d\n", i);

exit(0);

}

printf("setjmp returned 0 - good\n");

printf("calling inner...\n");

inner();

printf("inner returned - bad!\n");

}

***См. также***

[longjmp()](#f_longjmp)

***Возвращаемое значение:***

Функция **setjmp()** возвращает ноль после реального вызова, и ненулевое значение, если она, по-видимому, возвращается после вызова longjmp().

**SETUID**

***Синтаксис:***

**#include <cpm.h>**

**int setuid (unsigned char uid)**

***Описание:***

Функция **setuid()** устанавливает текущий код пользователя в **uid**. Значение **uid** должно быть числом в диапазоне 0-15.

***Пример:***

#include <stdio.h>

#include <cpm.h>

void main (void) {

setuid(2);

printf("Current user number is: %d\n", getuid());

}

***См. также***

[getuid()](#f_getuid)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**SETVBUF, SETBUF**

***Синтаксис:***

**#include <stdio.h>**

**int setvbuf (FILE \* stream, char \* buf, int mode, size\_t size)**

**void setbuf (FILE \* stream, char \* buf)**

***Описание:***

Функция **setvbuf()** позволяет изменить поведение буферизации потока stdio. Она заменяет функцию **setbuf()**, которая сохранена для обратной совместимости. Параметры **setvbuf()** следующие: **stream** определяет затронутый поток stdio. **buf** - указатель на буфер, который будет использоваться для всех последующих операций ввода-вывода в этот поток. Если **buf** равен NULL, то подпрограмма при необходимости выделит буфер из "кучи" размером BUFSIZ, как определено в <stdio.h>. Параметр **mode** может принять значения \_IONBF для полного отключения буферизации, \_IOFBF для полной буферизации или \_IOLBF для построчной буферизации. Полная буферизация означает, что связанный буфер будет сброшен, только когда заполнен, в то время как построчная буферизация означает, что буфер будет сброшен в конце каждой строки или когда будет требоваться ввод из другого потока stdio, **size** - размер предоставленного буфера. По умолчанию stdout и stdin являются буферизированными строками, когда они связаны с терминальным устройством и полностью буферизуются, когда связаны с файлом.

Если буфер предоставляется вызывающей стороной, то этот буфер останется связанным с этим потоком даже после вызовов fclose(), fopen() до изменения другим вызовов **setvbuf()**.

***Пример:***

#include <stdio.h>

char buffer[8192];

void main (void) {

int i, j;

/\* установить большой буфер для stdout \*/

setvbuf(stdout, buffer, \_IOFBF, sizeof buffer);

for(i = 0 ; i != 2000 ; i++)

if((i % 100) == 0)

printf("i = %4d\n", i);

else

for(j = 0 ; j != 1000 ; j++)

continue;

}

***См. также***

[fopen()](#f_fopen), [freopen()](#f_freopen), [fclose()](#f_fclose)

***Примечание:***

Если аргумент **buf** равен NULL, то **size** игнорируется.

Эта процедура не может использоваться в системе на основе ПЗУ.

**SET\_VECTOR**

***Синтаксис:***

**#include <intrpt.h>**

**isr set\_vector (isr \* vector, isr func)**

***Описание:***

Эта подпрограмма позволяет инициализировать вектор прерываний. Первый параметр должен быть адресом вектора прерывания (не номером вектора, а фактическим адресом) приведенным к указателю на **isr**, который является typedef указателем на функцию обработки прерываний. Второй параметр должен быть функцией, на которую будет указывать вектор прерывания. Он должен быть объявлен, используя спецификатор типа interrupt.

Не все компиляторы поддерживают эту процедуру. С некоторыми процессорами используются макросы ROM\_VECTOR(), RAM\_VECTOR() и CHANGE\_VECTOR(). Эти подпрограммы должны быть предпочтительными даже там, где поддерживается **set\_vector()**. См. файл **intrpt.h** или раздел руководства для конкретного процессора для определения, что поддерживается для конкретного компилятора.

В приведенном примере настраивается вектор для прерывания DOS Ctrl-break.

***Пример:***

#include <signal.h>

#include <stdlib.h>

#include <intrpt.h>

static far interrupt void brkintr (void) {

exit(-1);

}

#define BRKINT 0x23

#define BRKINTV ((far isr \*)(BRKINT \* 4))

void set\_trap (void) {

set\_vector(BRKINTV, brkintr);

}

***См. также***

[di()](#f_di), [ei()](#f_ei), [ROM\_VECTOR()](#f_ROM_VECTOR), [RAM\_VECTOR()](#f_RAM_VECTOR), [CHANGE\_VECTOR()](#f_CHANGE_VECTOR)

***Возвращаемое значение:***

Возвращаемое значение **set\_vector()** является предыдущим содержанием вектора, если **set\_vector()** реализуется как функция. Если она реализуется в виде макроса, у него нет возвращаемого значения.

***Примечание:***

Подпрограмма **set\_vector()** эквивалентна ROM\_VECTOR() и присутствует только для совместимости с компиляторами HI-TECH версий 5 и 6. Предполагается, что ROM\_VECTOR() будет использоваться вместо **set\_vector()** для максимальной совместимости с будущими версиями HI-TECH C.

***Типы данных:***

typedef interrupt void (\*isr)(void)

**SIGNAL**

***Синтаксис:***

**#include <signal.h>**

**void (\* signal (int sig, void (\*func)(int)))(int)**

***Описание:***

Функция **signal()** обеспечивает механизм для перехвата нажатий Ctrl-C (Ctrl-BREAK в MS-DOS) введенных в консоли во время ввода-вывода. В CP/M консоль опрашивается при каждом выполнении вызова ввода-вывода, в то время как в MS-DOS опрос зависит от установок команды BREAK. Если будет обнаружен Ctrl-C, то будет выполняться некоторое действие. Действие по умолчанию - немедленный выход. Оно может быть изменено с помощью **signal()**. Параметр **sig** функции signal может в настоящее иметь значение только SIGINT, означающее условие прерывания. Параметр **func** может иметь одно из значений - SIGDFL, представляющий действие по умолчанию, SIGIGN для полного игнорирования Ctrl-C или адрес функции, которая будет вызвана с одним параметром, номером пойманного сигнала, при обнаружении Ctrl-C. Поскольку единственным поддерживаемым сигналом является SIGINT, он всегда будет значением параметра вызванной функции.

***Пример:***

#include <signal.h>

#include <stdio.h>

#include <setjmp.h>

#include <stdlib.h>

jmp\_buf jb;

void far catch (int c) {

longjmp(jb, 1);

}

void main (void) {

int i;

if(setjmp(jb)) {

printf("\n\nCaught signal\n");

exit(0);

}

signal(SIGINT, catch);

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

printf("%6d\r", i);

}

}

***См. также***

[exit()](#f_exit)

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**SIN**

***Синтаксис:***

**#include <math.h>**

**double sin (double f)**

***Описание:***

Эта функция возвращает синус своего аргумента.

***Пример:***

#include <math.h>

#include <stdio.h>

#define C 3.141592/180.0

void main (void) {

double i;

for(i = 0 ; i <= 180.0 ; i += 10)

printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i\*C), cos(i\*C));

}

***См. также***

[cos()](#f_cos), [tan()](#f_tan), [asin()](#f_asin), [acos()](#f_cos), [atan()](#f_atan), [atan2()](#f_atan2)

***Возвращаемое значение:***

Значение синуса **f**.

**SPRINTF, VSPRINTF**

***Синтаксис:***

**#include <stdio.h>**

**int sprintf (char \* buf, const char \* fmt, ...)**

**#include <stdio.h>**

**#include <stdarg.h>**

**int vsprintf (char \* buf, const char \* fmt, va\_list ap)**

***Описание:***

Функция **sprintf()** действует аналогично printf(), за исключением того, что вместо помещения преобразованных выходных данных в поток stdout, символы помещаются в буфер **buf**. Результирующая строка будет завершена 0, и будет возвращено число символов в буфере.

Функция **vsprintf()** принимает в качестве параметра указатель, а не список параметров. См. описание [va\_start()](#f_va_start) для получения дополнительной информации о переменных списках аргументов.

***См. также***

[printf()](#f_printf), [fprintf()](#f_fprintf), [sscanf()](#f_sscanf)

***Возвращаемое значение:***

Обе эти функции возвращают количество символов, помещенных в буфер.

**SQRT**

***Синтаксис:***

**#include <math.h>**

**double sqrt (double f)**

***Описание:***

Эта функция возвращает синус своего аргумента.

***Пример:***

#include <math.h>

#include <stdio.h>

void main (void) {

double i;

for(i = 0 ; i <= 20.0 ; i += 1.0)

printf("square root of %.1f = %f\n", i, sqrt(i));

}

***См. также***

[exp()](#f_exp)

***Возвращаемое значение:***

Возвращает значение квадратного корня.

***Примечание:***

Если аргумент отрицательный, возникает ошибка выхода за пределы области допустимых значений.

**SRAND**

***Синтаксис:***

**#include <stdlib.h>**

**void srand (unsigned int seed)**

***Описание:***

Функция **srand()** инициализирует генератор случайных чисел, к которому обращается rand() с заданным началом. Это обеспечивает механизм изменения начальной точки псевдослучайной последовательности, заданной rand(). На z80 хорошим способом получения по настоящему случайного числа является обновленное значение регистра. В противном случае можно задействовать время ответа от консоли, или просто использовать системное время.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

#include <time.h>

void main (void) {

time\_t toc;

int i;

time(&toc);

srand((int)toc);

for(i = 0 ; i != 10 ; i++)

printf("%d\t", rand());

putchar('\n');

}

***См. также***

[rand()](#f_rand)

**SSCANF, VSSCANF**

***Синтаксис:***

**#include <stdio.h>**

**int sscanf (const char \* buf, const char \* fmt, ...)**

**#include <stdio.h>**

**#include <stdarg.h>**

**int vsscanf (const char \* buf, const char \* fmt, va\_list ap)**

***Описание:***

Функция **sscanf()** работает аналогично scanf(), за исключением того, что вместо преобразований, принимаемых из stdin, они берутся из строки в **buf**.

Функция **vsscanf()** принимает указатель на параметры, а не список параметров. См. описание [va\_start()](#f_va_start) для получения дополнительной информации о переменных списках аргументов.

***См. также***

[scanf()](#f_scanf), [fscanf()](#f_fscanf), [sprintf()](#f_sprintf)

***Возвращаемое значение:***

Возвращает значение EOF, если происходит сбой ввода, иначе возвращает число введенных элементов.

**STAT**

***Синтаксис:***

**#include <stat.h>**

**int stat (char \* name, struct stat \* statbuf)**

***Описание:***

Эта подпрограмма возвращает информацию о файле с именем **name**. Возвращаемые сведения зависят от операционной системы, но могут включать атрибуты файла (например, только для чтения), размер файла в байтах и время изменения файла и/или время доступа. Параметр **name** должен быть именем файла и может включать пути в DOS, коды пользователя в CP/M, и т.д. Параметр **statbuf** должен быть адресом структуры определенной в файле **stat.h**, которая будет заполнена информацией о файле. Строение структуры **stat** следующее:

{

short st\_mode; /\* Флаги \*/

long st\_atime; /\* Время доступа \*/

long st\_mtime; /\* Время изменения \*/

long st\_size; /\* Размер файла \*/

};

Время доступа и время изменения (в DOS они оба установлены во время изменения) находятся в секундах с 00:00:00 1 января 1970. Может использоваться функция ctime() для преобразования его в читаемое значение. Размер файла не требует пояснений.

Биты флагов следующие:

|  |  |
| --- | --- |
| **Флаг** | **Значение** |
| **S\_IFMT** | Маска типа файла |
| **S\_IFDIR** | Файл является каталогом |
| **S\_IFREG** | Обычный файл |
| **S\_IREAD** | Читаемый файл |
| **S\_IWRITE** | Записываемый файл |
| **S\_IEXEC** | Выполняемый файл |
| **S\_HIDDEN** | Скрытый файл |
| **S\_SYSTEM** | Системный файл |
| **S\_ARCHIVE** | Файл был записан в |

***Пример:***

#include <stdio.h>

#include <stat.h>

#include <time.h>

#include <stdlib.h>

void main (int argc, char \*\* argv) {

struct stat sb;

if(argc > 1) {

if(stat(argv[1], &sb)) {

perror(argv[1]);

exit(1);

}

printf("%s: %ld bytes, modified %s", argv[1],

sb.st\_size, ctime(&sb.st\_mtime));

}

exit(0);

}

***См. также***

[ctime()](#f_ctime), [creat()](#f_creat), [chmod()](#f_chmod)

***Возвращаемое значение:***

Функция **stat()** возвращает 0 в случае успеха, -1 в случае неудачи, например, если файл не может быть найден.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

***Типы данных:***

struct stat {

unsigned long t\_ino; /\* Не используется \*/

unsigned short st\_dev; /\* Не используется \*/

unsigned short st\_mode; /\* Флаги \*/

long st\_atime; /\* Время доступа \*/

long st\_mtime; /\* Время изменения \*/

long st\_size; /\* Размер файла в байтах \*/

}

**STRCAT**

***Синтаксис:***

**#include <string.h>**

**char \* strcat (char \* s1, const char \* s2)**

***Описание:***

Эта функция добавляет (соединяет) строку **s2** в конец строки **s1**. Результатом будет строка завершенная нулем. Аргумент **s1** должен указывать на массив символов, достаточно большой для хранения результирующей строки.

***Пример:***

#include <string.h>

#include <stdio.h>

void main (void) {

char buffer[256];

char \* s1, \* s2;

strcpy(buffer, "Start of line");

s1 = buffer;

s2 = " ... end of line";

strcat(s1, s2);

printf("Length = %d\n", strlen(buffer));

printf("string = \"%s\"\n", buffer);

}

***См. также***

[strcpy()](#f_strcpy), [strcmp()](#f_strcmp), [strncat()](#f_strncat), [strlen()](#f_strlen)

***Возвращаемое значение:***

Возвращается значение **s1**.

**STRCHR, STRICHR**

***Синтаксис:***

**#include <string.h>**

**char \* strchr (const char \* s, int c)**

**char \* strichr (const char \* s, int c)**

***Описание:***

Функция **strchr()** ищет в строке **s** появление символа **c**. Если он найден, возвращается указатель на этот символ, иначе возвращается NULL.

Функция **strichr()** является нечувствительной к регистру версией этой функции.

***Пример:***

#include <string.h>

#include <stdio.h>

void main (void) {

static char temp[] = "Here it is...";

char c = ’s’;

if(strchr(temp, c))

printf("Character %c was found in string\n", c);

else

printf("No character was found in string");

}

***См. также***

[strrchr()](#f_strrchr), [strlen()](#f_strlen), [strcmp()](#f_strcmp)

***Возвращаемое значение:***

Указатель на первое найденное соответствие, или NULL, если символ в строке не существует.

***Примечание:***

Хотя функция принимает целочисленный аргумент для символа, используются только младшие 8 бит значения.

**STRCMP, STRICMP**

***Синтаксис:***

**#include <string.h>**

**int strcmp (const char \* s1, const char \* s2)**

**int stricmp (const char \* s1, const char \* s2)**

***Описание:***

Функция **strcmp()** сравнивает свои два аргумента, представляющих собой завершенные нулем строки и возвращает целое число со знаком, чтобы указать, является ли **s1** меньше, равно или больше чем **s2**. Сравнение осуществляется с помощью стандартной последовательности сортировки, которая соответствует последовательности символов ASCII.

Функция **stricmp()** является нечувствительной к регистру версией этой функции.

***Пример:***

#include <string.h>

#include <stdio.h>

void main (void) {

int i;

if((i = strcmp("ABC", "ABc")) < 0)

printf("ABC is less than ABc\n");

else if(i >0)

printf("ABC is greater than ABc\n");

else

printf("ABC is equal to ABc\n");

}

***См. также***

[strlen()](#f_strlen), [strncmp()](#f_strncmp), [strcpy()](#f_strcpy), [strcat()](#f_strcat)

***Возвращаемое значение:***

Целое число со знаком меньше, равно или больше нуля.

***Примечание:***

Другие реализации C могут использовать другую последовательность сортировки. Возвращаемое значение отрицательное, ноль или положительное, т.е. не тестируйте явно на соответствие минус единице (-1) или единице (1).

**STRCPY**

***Синтаксис:***

**#include <string.h>**

**char \* strcpy (char \* s1, const char \* s2)**

***Описание:***

Эта функция копирует строку **s2**, завершенную нулем, в символьный массив, на который указывает **s1**. Массив назначения должен быть достаточно большим, чтобы вместить всю строку, включая нулевой символ конца строки.

***Пример:***

#include <string.h>

#include <stdio.h>

void main (void) {

char buffer[256];

char \* s1, \* s2;

strcpy(buffer, "Start of line");

s1 = buffer;

s2 = " ... end of line";

strcat(s1, s2);

printf("Length = %d\n", strlen(buffer));

printf("string = \"%s\"\n", buffer);

}

***См. также***

[strncpy()](#f_strncpy), [strlen()](#f_strlen), [strcat()](#f_strcat), [strlen()](#f_strlen)

***Возвращаемое значение:***

Возвращается указатель буфера назначения **s1**.

**STRCSPN**

***Синтаксис:***

**#include <string.h>**

**size\_t strcspn (const char \* s1, const char \* s2)**

***Описание:***

Функция **strcspn()** возвращает длину начального сегмента строки, на которую указывает **s1**, состоящая из символов, не из строки, на которую указывает **s2**.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

static char set[] = "xyz";

printf("%d\n", strcspn( "abcdevwxyz", set));

printf("%d\n", strcspn( "xxxbcadefs", set));

printf("%d\n", strcspn( "1234567890", set));

}

***См. также***

[strspn()](#f_strcspn)

***Возвращаемое значение:***

Возвращает длину сегмента.

**STRDUP**

***Синтаксис:***

**#include <string.h>**

**char \* strdup (const char \* s1)**

***Описание:***

Функция **strdup()** возвращает указатель на новую строку, которая является копией строки, на которую указывает **s1**. Пространство для новой строки получается с помощью malloc(). Если новая строка не может быть создана, возвращается нулевой указатель.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

char \* ptr;

ptr = strdup("This is a copy");

printf("%s\n", ptr);

}

***Возвращаемое значение:***

Указатель на новую строку или NULL, если новая строка не может быть создана.

**STRLEN**

***Синтаксис:***

**#include <string.h>**

**size\_t strlen (const char \* s)**

***Описание:***

Функция **strlen()** возвращает количество символов в строке **s**, не включая нулевой разделитель.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

char buffer[256];

char \* s1, \* s2;

strcpy(buffer, "Start of line");

s1 = buffer;

s2 = " ... end of line";

strcat(s1, s2);

printf("Length = %d\n", strlen(buffer));

printf("string = \"%s\"\n", buffer);

}

***Возвращаемое значение:***

Количество символов, предшествующих нулевому разделителю.

**STRNCAT**

***Синтаксис:***

**#include <string.h>**

**char \* strncat (char \* s1, const char \* s2, size\_t n)**

***Описание:***

Эта функция добавляет (соединяет) строку **s2** в конец строки **s1**. Будут скопированы не более **n** символов, и результат будет завершен нулем. Аргумент **s1** должен указывать на массив символов, достаточно большой для хранения результирующей строки.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

char buffer[256];

char \* s1, \* s2;

strcpy(buffer, "Start of line");

s1 = buffer;

s2 = " ... end of line";

strncat(s1, s2);

printf("Length = %d\n", strlen(buffer));

printf("string = \"%s\"\n", buffer);

}

***См. также***

[strcpy()](#f_strcpy), [strcmp()](#f_strcmp), [strcat()](#f_strcat), [strlen()](#f_strlen)

***Возвращаемое значение:***

Возвращается значение **s1**.

**STRNCMP, STRNICMP**

***Синтаксис:***

**#include <string.h>**

**int strncmp (const char \* s1, const char \* s2, size\_t n)**

**int strnicmp (const char \* s1, const char \* s2, size\_t n)**

***Описание:***

Функция **strncmp()** сравнивает свои два аргумента, представляющих собой завершенные нулем строки максимум до **n** символов и возвращает целое число со знаком, чтобы указать, является ли **s1** меньше, равно или больше чем **s2**. Сравнение осуществляется с помощью стандартной последовательности сортировки, которая соответствует последовательности символов ASCII.

Функция **strnicmp()** является нечувствительной к регистру версией этой функции.

***Пример:***

#include <string.h>

#include <stdio.h>

void main (void) {

int i;

i = strncmp("abcxyz", "abcxyz", 5);

if(i == 0)

printf("Both strings are equal\n");

else if(i > 0)

printf("String 2 less than string 1\n");

else

printf("String 2 is greater than string 1\n");

}

***См. также***

[strlen()](#f_strlen), [strcmp()](#f_strcmp), [strcpy()](#f_strcpy), [strcat()](#f_strcat)

***Возвращаемое значение:***

Целое число со знаком меньше, равно или больше нуля.

***Примечание:***

Другие реализации C могут использовать другую последовательность сортировки. Возвращаемое значение отрицательное, ноль или положительное, т.е. не тестируйте явно на соответствие минус единице (-1) или единице (1).

**STRNCPY**

***Синтаксис:***

**#include <string.h>**

**char \* strncpy (char \* s1, const char \* s2, size\_t n)**

***Описание:***

Эта функция копирует строку **s2**, завершенную нулем, в символьный массив, на который указывает **s1**. В лучшем случае копируются **n** символов. Если строка **s2** длиннее **n**, то строка назначения не будет завершена нулем. Массив назначения должен быть достаточно большим, чтобы содержать всю строку, включая нулевой разделитель.

***Пример:***

#include <string.h>

#include <stdio.h>

void main (void) {

char buffer[256];

char \* s1, \* s2;

strcpy(buffer, "Start of line", 6);

s1 = buffer;

s2 = " ... end of line";

strcat(s1, s2);

printf("Length = %d\n", strlen(buffer));

printf("string = \"%s\"\n", buffer);

}

***См. также***

[strcpy()](#f_strcpy), [strcat()](#f_strcat), [strlen()](#f_strlen), [strcmp()](#f_strcmp)

***Возвращаемое значение:***

Возвращается указатель буфера назначения **s1**.

**STRPBRK**

***Синтаксис:***

**#include <string.h>**

**char \* strpbrk (const char \* s1, const char \* s2)**

***Описание:***

Функция **strpbrk()** возвращает указатель на первое возникновение в строке **s1** любого символа из представленных в виде строки **s2** или нулевого указателя, если никакой символ из **s2** не существует в **s1**.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

char \* str = "This is a string.";

while(str != NULL) {

printf( "%s\n", str );

str = strpbrk( str+1, "aeiou" );

}

}

***Возвращаемое значение:***

Указатель на первый соответствующий символ или NULL, если символ не найден.

**STRRCHR, STRRICHR**

***Синтаксис:***

**#include <string.h>**

**char \* strrchr (const char \* s, int c)**

**char \* strrichr (const char \* s, int c)**

***Описание:***

Функция **strrchr()** аналогична функции strchr(), но поиск выполняется с конца строки, а не с начала, т. е. находит последнее вхождение символа **c** в строке **s**, завершенной значением null. В случае успеха возвращается указатель на это вхождение, в противном случае возвращается значение NULL.

Функция **strrichr()** является нечувствительной к регистру версией этой функции.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

char \* str = "This is a string.";

while(str != NULL) {

printf( "%s\n", str );

str = strrchr( str+1, ’s’);

}

}

***См. также***

[strchr()](#f_strchr), [strlen()](#f_strlen), [strcmp()](#f_strcmp), [strcpy()](#f_strcpy), [strcat()](#f_strcat)

***Возвращаемое значение:***

Указатель на символ или NULL, если не найден.

**STRSPN**

***Синтаксис:***

**#include <string.h>**

**size\_t strspn (const char \* s1, const char \* s2)**

***Описание:***

Функция **strspn()** возвращает длину начального сегмента строки, на которую указывает **s1** который полностью состоит из символов строки, на которую указывает **s2**.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

printf("%d\n", strspn("This is a string", "This"));

printf("%d\n", strspn("This is a string", "this"));

}

***См. также***

[strcspn()](#f_strcspn)

***Возвращаемое значение:***

Длину сегмента.

**STRSTR, STRISTR**

***Синтаксис:***

**#include <string.h>**

**char \* strstr (const char \* s1, const char \* s2)**

**char \* stristr (const char \* s1, const char \* s2)**

***Описание:***

Функция **strstr()** находит первое вхождение последовательности символов в строке, на которую указывает **s2** в строке, на которую указывает **s1**.

Функция **stristr()** является нечувствительной к регистру версией этой функции.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

printf("%d\n", strstr("This is a string", "str"));

}

***Возвращаемое значение:***

Указатель на расположенную строку или нулевой указатель, если строка не найдена.

**STRTOK**

***Синтаксис:***

**#include <string.h>**

**char \* strtok (char \* s1, const char \* s2)**

***Описание:***

Ряд вызовов **strtok()** разбивает строку **s1** (которая состоит из последовательности из нуля или более текстовых лексем, разделенных одним или несколькими символами строки-разделителя **s2**) на свои отдельные лексемы.

Первый вызов должен указывать на строку **s1**. Этот вызов возвращает указатель на первый символ первой лексемы или значение NULL, если лексемы не найдены. Символ разделителя между лексемами замещается символом NULL, который завершает текущую лексему.

Для последующих вызовов **strtok()**, для **s1** должен быть установлен нулевой указатель. Эти вызовы начинают поиск с конца найденной лексемы и снова возвращают указатель на первый символ следующей лексемы или значение NULL, если последующие лексемы не найдены.

***Пример:***

#include <stdio.h>

#include <string.h>

void main (void) {

char \* ptr;

char \* buf = "This is a string of words.";

char \* sep\_tok = ".,?! ";

ptr = strtok(buf, sep\_tok);

while(ptr != NULL) {

printf("%s\n", ptr);

ptr = strtok(NULL, sep\_tok);

}

}

***Возвращаемое значение:***

Возвращает указатель на первый символ лексемы или нулевой указатель, если лексема не найдена.

***Примечание:***

Разделитель строки **s2** может отличаться от вызова к вызову.

**TAN**

***Синтаксис:***

**#include <math.h>**

**double tan (double f)**

***Описание:***

Функция **tan()** вычисляет тангенс **f**.

***Пример:***

#include <math.h>

#include <stdio.h>

#define C 3.141592/180.0

void main (void) {

double i;

for(i = 0 ; i <= 180.0 ; i += 10)

printf("tan(%3.0f) = %f\n", i, tan(i\*C));

}

***См. также***

[sin()](#f_sin), [cos()](#f_cos), [asin()](#f_asin), [acos()](#f_acos), [atan()](#f_atan), [atan2()](#f_atan2)

***Возвращаемое значение:***

Тангенс **f**.

**TIME**

***Синтаксис:***

**#include <time.h>**

**time\_t time (time\_t \* t)**

***Описание:***

Эта функция не предусмотрена, поскольку она зависит от целевой системы, предоставляющей текущее время. Эта функция должна быть реализована пользователем. При реализации эта функция должна возвращать текущее время в секундах с 00:00:00 1 января 1970 года. Если аргумент **t** не равен NULL, то такое же значение сохраняется в объекте, на который указывает **t**.

***Пример:***

#include <stdio.h>

#include <time.h>

void main (void) {

time\_t clock;

time(&clock);

printf("%s", ctime(&clock));

}

***См. также***

[ctime()](#f_ctime), [gmtime()](#f_gmtime), [localtime()](#f_localtime), [asctime()](#f_astime)

***Возвращаемое значение:***

Эта процедура, если реализована, вернет текущее время в секундах с 00:00:00 1 января 1970 года.

***Примечание:***

Процедура **time()** не предоставляется, если требуется, то пользователь должен будет реализовать эту процедуру в соответствии со спецификациями, описанными выше.

**TMPFILE**

***Синтаксис:***

**#include <stdio.h>**

**FILE \* tmpfile (void)**

***Описание:***

Эта функция создает временный двоичный файл, который автоматически закрывается и удаляется при завершении программы.

***См. также***

[fopen()](#f_fopen)

***Возвращаемое значение:***

Указатель на созданный файловый поток. Если временный файл не может быть открыт, возвращается нулевой указатель.

***Примечание:***

Файл открыт, используя режим "wb+".

Эта процедура не может использоваться в системе на основе ПЗУ.

**TMPNAM**

***Синтаксис:***

**#inlcude <stdio.h>**

**#include <stdlib.h>**

**char \* tmpnam (char \* s)**

***Описание:***

Функция **tmpnam()** создает допустимое имя файла, отличное от имени существующих файлов. Функция **tmpnam()** может быть вызвана до TMP\_MAX раз, при этом каждый раз создается другая строка. Поведение **tmpnam()** не определено для каких-либо дальнейших обращений к ней.

***Пример:***

#include <stdio.h>

#include <stdlib.h>

void main (void) {

char \* filename;

FILE \* fp;

filename = tmpnam(NULL);

if(filename != NULL) {

fp = fopen(filename, "wt");

/\* Может использоваться для создания отчета об ошибках \*/

fclose(fp);

free(filename);

else

printf("Cannot create temp file name\n");

}

***Возвращаемое значение:***

Если аргумент является нулевым указателем, **tmpnam()** оставляет результат во внутреннем статическом объекте и возвращает указатель на этот объект. Обратите внимание, что последующие вызовы **tmpnam()** могут изменить этот объект. Если аргумент не является нулевым указателем, предполагается, что он указывает на символ не менее **L\_tmpnam** символов. В этом случае результат записывается в массив, и возвращается аргумент.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**TOLOWER, TOUPPER, TOASCII**

***Синтаксис:***

**#include <ctype.h>**

**char toupper (int c)**

**char tolower (int c)**

**char toascii (int c)**

***Описание:***

Функция **toupper()** преобразует строчный аргумент нижнего регистра в верхний регистр, **tolower()** выполняет обратное преобразование, а макрос **toascii()** возвращает результат, гарантированный в диапазоне 0-0177. Функции **toupper()** и **tolower()** возвращают свои аргументы, если они не являются буквенными символами.

***Пример:***

#include <stdio.h>

#include <ctype.h>

#include <string.h>

void main (void) {

char \* array1 = "aBcDE";

int i;

for(i=0;i < strlen(array1); ++i) {

printf("%c", tolower(array1[i]));

}

printf("\n");

}

***См. также***

[islower()](#f_islower), [isupper()](#f_isupper), [isascii()](#f_isascii) и др.

**UNGETC**

***Синтаксис:***

**#include <stdio.h>**

**int ungetc (int c, FILE \* stream)**

***Описание:***

Функция **ungetc()** попытается вернуть символ **c** в именованный поток, так что последующий вызов операции getc() вернет символ. Если поток не буферизован, будет разрешен максимум один уровень отката, даже это может быть невозможно. Если функция **ungetc()** не может быть выполнена, возвращается EOF.

***Пример:***

#include <stdio.h>

#include <ctype.h>

void main (void) {

FILE \* stream;

int c;

long number = 0;

if(stream = fopen("temp.dat", "r")) {

c = fgetc(stream);

while(isdigit(c)) {

number = number\*10 + (c - ’0’);

c = fgetc(stream);

}

ungetc(c, stream);

printf("Read number is = %ld\n", number);

fclose(stream);

}

else

printf("Could not open file.\n");

}

***См. также***

[getc()](#f_getc)

***Возвращаемое значение:***

Возвращает символ, пододвинутый обратно, или EOF, если **ungetc()** не может быть выполнена.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**UNGETCH**

***Синтаксис:***

**#include <conio.h>**

**void ungetch (char c)**

***Описание:***

Функция **ungetch()** продвинет символ c обратно в консольный поток, так что последующая операция getch() вернет символ. Разрешено продвижение назад не более одного уровня.

***См. также***

[getch()](#f_getch), [getche()](#f_getche)

**UNLINK**

***Синтаксис:***

**#include <unixio.h>**

**int unlink (const char \* name)**

***Описание:***

Функция **unlink()** удалит (сотрет) именованный файл, то есть удалит файл из его каталога. См. [open()](#f_open) для описания конструкции имени файла. В случае успеха будет возвращен ноль, минус единица, если файл не существует или не может быть удален. Функция ANSI remove() предпочтительнее **unlink()**.

***Пример:***

#include <unixio.h>

void main (void) {

if(unlink("test.fil") < 0)

perror("test.fil");

}

***См. также***

[open()](#f_open), [close()](#f_close), [rename()](#f_rename), [remove()](#f_remove)

***Возвращаемое значение:***

В случае успеха возвращается ноль, минус единица (-1), если файл не существует, или он не может быть удален.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**VA\_START, VA\_ARG, VA\_END**

***Синтаксис:***

**#include <stdarg.h>**

**void va\_start (va\_list ap, parmN)**

**type va\_arg (ap, type)**

**void va\_end (va\_list ap)**

***Описание:***

Эти макросы предоставляются для обеспечения доступа переносимым способом к параметрам функции, представленных в прототипе символом многоточие (...), где тип и число параметров, предоставленных функции, не известны во время компиляции.

Самый правый параметр функции (обозначенный **parmN**) играет важную роль в этих макросах, так как он является начальной точкой для доступа к последующим параметрам. В функции, принимающей переменное число параметров, должна быть объявлена переменная типа **va\_list**, затем вызывается макрос **va\_start()** с этой переменной и именем **parmN**. Он инициализирует переменную, чтобы позволить последующим вызовам макроса **va\_arg()** получать доступ к последовательным параметрам.

Каждый вызов **va\_arg()** требует два параметра. Ранее определенную переменную и имя типа, следующего параметра. Обратите внимание, что любые параметры доступные таким образом будут расширены в соответствии с соглашениями по умолчанию к int, unsigned int или double. Например, если передан символьный параметр, к нему должен получить доступ va\_arg(ap, int), так как символ будет расширен до int.

Ниже приведен пример функции, принимающей один целочисленный параметр, сопровождаемый многими другими параметрами. В этом примере функция ожидает, что последующие параметры будут указателями на char, но отметьте, что компилятор не знает об этом, и программисты обязаны гарантировать, что предоставлены корректные параметры.

***Пример:***

#include <stdio.h>

#include <stdarg.h>

void pf (int a, ...) {

va\_list ap;

va\_start(ap, a);

while(a--)

puts(va\_arg(ap, char \*));

va\_end(ap);

}

void main (void) {

pf(3, "Line 1", "line 2", "line 3");

}

**WRITE**

***Синтаксис:***

**#include <unixio.h>**

**size\_t write (int fd, const void \* buf, size\_t cnt)**

***Описание:***

Функция **write()** записывает из буфера **buf** до **cnt** байтов в файл, связанный с дескриптором файла **fd**. Возвращается число фактически записанных байтов. При ошибке возвращается EOF или значение меньше чем **cnt**. В любом случае любое возвращаемое значение, не равное **cnt**, должно быть обработано как ошибка (см. [read()](#f_read)).

***Пример:***

#include <unixio.h>

void main (void) {

write(1, "A test string\r\n", 15);

}

***См. также***

[open()](#f_open), [close()](#f_close), [read()](#f_read)

***Возвращаемое значение:***

Возвращается количество фактически записанных байтов. При ошибке возвращается EOF или значение меньше, чем **cnt**.

***Примечание:***

Эта процедура не может использоваться в системе на основе ПЗУ.

**XTOI**

***Синтаксис:***

**#include <stdlib.h>**

**unsigned xtoi (const char \* s)**

***Описание:***

Функция **xtoi()** сканирует переданную ей символьную строку, пропуская начальные пробелы, читая необязательный знак, и преобразует ASCII представление шестнадцатеричного числа в целое число.

***Пример:***

#include <stdlib.h>

#include <stdio.h>

void main (void) {

char buf[80];

int i;

gets(buf);

gets(buf);

i = xtoi(buf);

printf("Read %s: converted to %x\n", buf, i);

}

***См. также***

[atoi()](#f_atoi)

***Возвращаемое значение:***

Целое число со знаком. Если в строке число не найдено, будет возвращен ноль.

**\_GETARGS**

***Синтаксис:***

**#include <sys.h>**

**char \*\* \_getargs (char \* buf, char \* name)**

**extern int \_argc\_**

***Описание:***

Эта подпрограмма выполняет перенаправление ввода-вывода (только в CP/M) и подстановочное расширение. В MS-DOS перенаправление ввода-вывода выполняется операционной системой. Она вызывается из кода запуска, чтобы воздействовать на командную строку, если для команды C используется параметр -R, но также может быть вызвана из кода, написанного пользователем. Если параметр **buf** равен нулю, она читает строки текста из стандартного ввода. Если стандартный ввод будет терминалом (обычно консоль), то параметр **name** будет записываться в стандартный поток ошибок как подсказка. Если параметр **buf** будет не ноль, то он будет использоваться в качестве источника строки для обработки. Возвращенное значение - указатель на массив строк, точно как был бы указан параметр argv функции main(). Число строк в массиве может быть получено из глобальной переменной **\_argc\_**.

Для каждого слова в обработанном буфере будет одна строка в массиве. Для включения пробелов в слова могут использоваться одинарные (') или двойные (") кавычки. Если какие-либо подстановочные символы (? или \*), появляются в слове без кавычек, оно будет расширено в строку слов, по одному для каждого файла, соответствующего слову. Для этого расширения действуют обычные соглашения CP/M и DOS. В CP/M любые вхождения символов перенаправления > и < вне кавычек обрабатываются следующим образом:

**> name** стандартный вывод будет перенаправлен в файл с именем **name**.

**< name** стандартный ввод будет перенаправлен из файл с именем **name**.

**>> name** стандартный вывод будет добавляться в файл с именем **name**.

Между символами > или < и именем файла пробельные символы необязательны, однако символ перенаправления, за которым не следует имя файла, является ошибкой. Также возникает ошибка, если файл не может быть открыт для ввода или создан для вывода. Добавление перенаправления (>>) создаст файл, если он не существует. Если источником обрабатываемого текста является стандартный ввод, могут быть предоставлены несколько строк, заканчивая каждую строку (кроме последней) символом обратной косой черты (\). Он служит символом продолжения. Обратите внимание, что символ новой строки после обратной косой черты игнорируется и не обрабатывается как пробел.

***Пример:***

#include <sys.h>

void main (int argc, char \*\* argv) {

extern char \*\* \_getargs(char \*, char \*);

extern int \_argc\_;

if(argc == 1) { /\* аргументы отсутствуют \*/

argv = \_getargs(0, "myname");

argc = \_argc\_;

}

.

.

.

***Возвращаемое значение:***

Указатель на массив строк.

***Примечание:***

В CP/M первый элемент возвращаемого массива не будет именем программы, но будет аргументом имени. Эта процедура не может использоваться в системе на основе ПЗУ.

# Предметный указатель

\_

**\_getargs()** 335

#

#pragma

jis 134

nojis 134

printf\_check 134

psect 46, 134

psect strings 135

switch 136

А

Автоматические переменные 36, 127

Адрес

загрузки 36, 156

загрузки psect 132

сегмента 159

сегмента (8086) 156

ссылки 36, 156

ссылки psect 132

Ассемблер

временная метка 142

выражения 143

идентификаторы 141

метка 142

набор символов 140

символьная константа 141

спецификатор основания 140

строки 143

Б

Библиотека 166

Библиотечные функции 25

В

Включаемые файлы 22

Внутренние приложения компилятора 22

Внутрисхемный эмулятор 20

Временные переменные 36

Временные файлы 22

Входные файлы

ассемблера 24

библиотечные 24

заголовков 22

исходные файлы C 22

объектные 24

Выбор режима прерывания 112

Вывод препроцессора 25

Выпадающие меню 52

Выходные данные препроцессора 26

Г

Генератор кода 27

Глобальная переменная

**environ** 282

**errno** 298

Глобальная psect 34, 156

Глобальные символы 27

Горячие клавиши 53

Граф вызовов 164

Группировка и перемещение psect 32

Д

Диалоговое окно

INSTALL 13

Директива ассемблера

CLASS 156

END 146

GLOBAL 146

IPR 151

IPRC 151

LOCAL 142, 150

MACRO 149

ORG 37, 148

PSECT 146

REPT 151

SIGNAT 152

Директива препроцессора

include 22

Директивы

#asm 118

#endasm 118

компилятора 133

asm() 118

psect 29, 34

Директивы условного ассемблирования

COND 149

ELSE 149

ENDC 149

IF 149

Драйвер

командной строки 16, 21

HPD 21

Драйвер компилятора 15

И

Изменение чувствительности мыши 56

Имя выходного файла компоновщика 161

Имя сегмента 46

Имя текущего файла 62

Имя файла листинга 76

Интегрированная среда разработки 49

Информационное окно INSTALL 15

К

Кадр стека 121

Квалификатор

*interrupt* 111

Квалификаторы типа

code 108, 109

const 108

persistent 108, 109

port 106

volatile 108

Классификатор функции

nmi 117

Классы 41

определение 41

правила использования 42

Ключ установки 14

Ключевое слово

interrupt 111

persistent 19

reentrant 207

Код

ассемблерный 29

времени выполнения 130, 132

запуска времени выполнения 129

запуска среды выполнения 35

Комментарии стиля C++ 66

Компилятор 21

Компиляция 21

Компоновщик 31, 120, 155

Контрольная сумма 32

Куча 127

Л

Локальные psect 156

М

Макросы

CHANGE\_VECTOR 113

di() 112

ei() 112

im() 112, 114

RAM\_VECTOR 113

READ\_RAM\_VECTOR 113

ROM\_VECTOR 112

Маркеры неразрешенных адресов 30

Машинный код 29

микро-программы в ПЗУ 109

Модуль трансляции 25

Н

Наложение psect 36

Начальный адрес программы 129

Непредвиденное расширение знака 213

О

Область содержимого редактора 62

Объектные файлы 159

абсолютные 29

перемещаемые 29

периода выполнения 32

Окно сообщений

HPDZ 49

INSTALL 13

Окно состояния программы INSTALL 13

Оконный интерфейс HI-TECH 49

Определение векторов прерываний 37

Оптимизация

глобальная 29

локальная 29

Опции

ассемблера 139

библиотекаря 167

компоновщика 157

CREF 170

objtohex 169

Отладчик LUCIFER 73, 136, 173, 281, 303

П

Параметр командной строки

HPDZ 49

Параметры функции 36

Переменная окружения DOS

TEMP 13, 15, 22

Подпрограмма

set\_vector() 114

Подсветка синтаксиса C 72

Полоса прокрутки 56

Преобразователь форматов 32

Препроцессор 25

Привязка адресов 32

Приложение

clist 25

cpp 25

CREF 170

cromwell 32

DUMP 30

hlink 31, 163

HPDZ 49

INSTALL 13

LIBR (библиотекарь) 166

objtohex 32, 169

p1 26

ZAS (ассемблер) 138

zc 15

Проверка сигнатуры 118

Программатор ППЗУ 20

Программные секции psect 34

Промежуточные группировки psect 45

Простой тип 231

Процедура

powerup 130

Псевдо-оператор

DEFB, DB 148

DEFF 149

DEFL 148

DEFS, DS 149

DEFW 148

EQU 148

GLOBAL 118

SIGNAT 118, 120

Р

Рабочий каталог 57

Разделы программы 29

Размер табуляции 71

Размеры окна 55

Рамка редактора 62

Расположение объектов по абсолютному адресу 46

Расположение секций psect 36

Расширение файла

as 24, 28

bat 21

c 22

h 22

ini 21, 22

lib 24

obj 24

prj 21, 50

rlf 30

sdb 29

sym 32

Режим автоматического отступа 63

Режим вставки 63

Режим отображения экрана 50

Режим C 63

Режимы прерываний

0 114

1 114

2 116

С

Сегмент 45, 159

Селектор сегмента 159

Серийный номер 14

Сигнатура 119

Синтаксический анализатор 26

Системное меню 56

Слияние секций psect 36

Сообщение об авторском праве 133

Стандартные цветовые схемы 51

Статические переменные 127

Строка меню

HPDZ 49

Структура

**div\_t** 263

**ldiv\_t** 289

stat 319

**tm** 248

Сцепки 45

Т

Текстовый редактор

EDIT 18

HPDZ 49

Тип процессора 162

У

Указатель стека 127

Уровень предупреждений 134, 163

Ф

Файл

библиотечный 32

заголовочный 22

инициализации 21

информации микросхем 21

исполняемый 26

листинга ассемблера 30

листинга C 25

объектный 29

отладки 29

проекта 21

рапределения памяти 164

распределения памяти 32

с перемещаемым листингом 30

символов 32

AUTOEXEC.BAT 15

CONFIG.SYS 15

HIMEM.SYS 13

HPDZ.INI 50

ser180.c 136

sersio.c 136

z180.h 117

Флаг директивы ассемблера

ABS 147

CLASS 147

GLOBAL 147

LOCAL 147

OVRLD 147

PURE 147

RELOC 147

SELECTOR 147

SIZE 147

Флаги директивы psect

ABS 37

CLASS 38

GLOBAL 36

LOCAL 36

OVLRD 36

RELOC 37

Формат библиотеки 167

Формат выходного файла

бинарный 20

двоичный 32

стандартный 32

American Automation HEX 20

Intel HEX 20

Motorola S1/S9 20

Tektronix HEX 20

Ubrof 20

Формат файла распределения памяти 161

Функция 126

обработки прерываний 111

basenear 126

fast interrupt 117

main() 129

sbrk() 127

Э

Элемент управления ассемблером

\*EJECT 153

\*HEADING 153

\*INCLUDE 153

\*LIST 153

\*TITLE 154

A

**abort()** 246

**abs()** 247

**acos()** 247

**asctime()** 248

**asin()** 249

**assert()** 249

**atan()** 250

**atan2()** 250

**atexit()** 251

**atof()** 251

**atoi()** 252

**atol()** 253

B

**bdos()** 253

**bdoshl()** 254

**bios()** 254

**bsearch()** 255

C

Cредства обработки ошибок HPDZ 58

Cтрока состояния редактора 62

**calloc()** 256

**ceil()** 256

**cgets()** 257

**CHANGE\_VECTOR()** макрос 305

**chmod()** 257

**close()** 258

**clreof()**макрос 259

**clrerr()** макрос 259

**cos()** 259

**cosh()** 260

**cputs()** 260

**creat()** 261

**ctime()** 262

D

**di()** 262

**div()** 263

**dup()** 263

E

**ei()** 262

**eval\_poly()** 264

**execl()** 265

**execv()** 265

**exit()** 265

**exp()** 266

F

**fabs()** 266

**fclose()** 267

**fdopen()** 267

**feof()** макрос 268

**ferror()** макрос 268

**fflush()** 268

**fgetc()** 269

**fgets()** 270

**fileno()** макрос 270

**floor()** 272

**fopen()** 272

**fprintf()** 274

**fputc()** 275

**fputs()** 275

**fread()** 276

**free()** 276

**freopen()** 277

**frexp()** 277

**fscanf()** 278

**fseek()** 279

**ftell()** 280

**fwrite()** 280

G

**getc()** макрос 281

**getch()** 281

**getchar()** 282

**getche()** 281

**getenv()** 282

**gets()** 283

**getuid()** 284

**getw()** 284

**gmtime()** 285

H

heap 127, 129

I

**im()** 286

**isalnum()** макрос 286

**isalpha()** макрос 286

**isascii()** макрос 286

**isatty()** 287

**iscntrl()** макрос 286

**isdigit()** макрос 286

**isgraph()** макрос 286

**islower()** макрос 286

**isprint()** макрос 286

**ispunct()** макрос 286

**isspace()** макрос 286

**isupper()** макрос 286

K

**kbhit()** 288

L

**ldexp()** 288

**ldiv()** 289

**localtime()** 289

**log()** 290

**log10()** 290

**longjmp()** 291

**lseek()** 292

lvalue 211

M

**malloc()** 293

**memchr()** 293

**memcmp()** 294

**memcpy()** 295

**memmove()** 295

**memset()** 296

**modf()** 296

O

**open()** 297

P

**perror()** 298

**persist\_check()** 299

**persist\_validate()** 299

**pow()** 300

**pragma** 133

**printf()** 300

psect

basecode 128

baseram 128

bss 129, 132, 155

const 128

data 129, 155

heap 129

im2vecs 128

lowtext 128

ltext 128

nvram 129

ramstart 128

stack 129, 132

strings 128

text 128, 155

vectors 127

**putc()** 302

**putch()** 303

**putchar()** 303

**puts()** 304

**putw()** 304

Q

**qsort()** 305

R

**RAM\_VECTOR()** макрос 305

**rand()** 306

**READ\_RAM\_VECTOR()** макрос 305

**read()** 307

**realloc()** 307

**remove()** 308

**rename()** 308

**rewind()** 309

**ROM\_VECTOR()** макрос 310

run-time 35

S

**sbrk()** 310

**scanf()** 311

**set\_vector()** 314

**setbuf()** 313

**setjmp()** 312

**setuid()** 312

**setvbuf()** 313

**signal()** 315

**sin()** 316

**sinh()** 260

**sprintf()** 316

**sqrt()** 317

**srand()** 317

**sscanf()** 318

startoff 114, 128, 130, 132

startup 35

**stat()** 318

**strcat()** 320

**strchr()** 320

**strcmp()** 321

**strcpy()** 322

**strcspn()** 322

**strdup()** 323

**strichr()** 320

**stricmp()** 321

**stristr()** 327

**strlen()** 323

**strncat()** 324

**strncmp()** 324

**strncpy()** 325

**strnicmp()** 324

**strpbrk()** 326

**strrchr()** 326

**strrichr()** 326

**strspn()** 327

**strstr()** 327

**strtok()** 328

T

**tan()** 328

**tanh()** 260

**time()** 329

**tmpfile()** 329

**tmpnam()** 330

**toascii()** макрос 331

**tolower()** 331

**toupper()** 331

U

**ungetc()** 331

**ungetch()** 332

**unlink()** 332

V

**va\_arg()** макрос 333

**va\_end()** макрос 333

**va\_start()** макрос 333

**vfprintf()** 274

**vfscanf()** 278

**vprintf()** 301

**vscanf()** 311

**vsprintf()** 316

**vsscanf()** 318

W

WordStar 62

**write()** 334

X

**xtoi()** 334

1. Оригинал документа, использованный для этого перевода (Прим. перев.) [↑](#footnote-ref-1)
2. Драйвера командной строки и драйвера HPD имеют специфичные для процессора имена, такие как picc, c51 или HPDXA, HPDPIC и т.д. [↑](#footnote-ref-2)
3. Расширения, перечисленные в этих таблицах, находятся в нижнем регистре. Компиляторы DOS не различают заглавные и строчные имена файлов и расширений, но в интересах написания переносимых про грамм вам следует использовать строчные расширения имен файлов и ссылки на эти файлы в вашем коде, так как компиляторы UNIX обрабатывают регистр правильно. [↑](#footnote-ref-3)
4. В интересах ясности генерация этого файла на [Рисунке 2-1](#r21) на странице 19 не показана. [↑](#footnote-ref-4)
5. Ранние компоновщики HI-TECH назывались **link**. [↑](#footnote-ref-5)
6. Термин "ПЗУ" будет использован по отношению к любой энергонезависимой памяти. [↑](#footnote-ref-6)
7. Ассемблер не изменяет директивы psect кроме включение описаний каждой в объектный файл. [↑](#footnote-ref-7)
8. Опция **-A** компилятора PIC служит другой цели. Большинство устройств PIC имеют только внутреннюю память и таким образом, опция памяти не требуется компилятору. Высокопроизводительные PIC могут иметь внешнюю память, на это компилятору указывает опция **-ROM** в CLD или диалоговое окно **ROM** **&** **RAM** **addresses...** в HPDPIC. Опция **-A** используется для смещения всего образа ПЗУ, при использовании высокопроизводительных устройств. [↑](#footnote-ref-8)
9. Некоторые процессоры могут потребовать для выравнивания слов промежуток между кодом или данными. Эти промежутки могут быть обработаны компилятором, но здесь они не рассматриваются. [↑](#footnote-ref-9)
10. BSS означает '**B**lock **S**tarted by **S**ymbol', мнемоника из ассемблера IBM 7094 (Прим. перев.) [↑](#footnote-ref-10)
11. **HPD**, сокращение от англ. **H**I-TECH C **P**rogrammer’s **D**evelopment system (Прим. перев.) [↑](#footnote-ref-11)
12. Нажатие левой кнопки в рамке окна с полосой прокрутки имеет совершенно другой эффект, описанный далее в этой главе. [↑](#footnote-ref-12)
13. Баллистический порог мыши - это скорость, за которой ответ указателя на дальнейшее движение становится экспоненциальным. Некоторые примитивные драйверы мыши не поддерживают эту функцию. [↑](#footnote-ref-13)
14. Сокращение от английского - **U**niversal **B**inary **I**mage **R**elocatable **F**ormat (Прим. перев.) [↑](#footnote-ref-14)
15. NMI – сокращение от англ. **N**on **M**askable **I**nterrupts (Немаскируемые прерывания) (Прим. перев.) [↑](#footnote-ref-15)
16. JIS – сокр. от англ. **J**apan **I**ndustrial **S**tandards (японские промышленные стандарты). (Прим. перев.) [↑](#footnote-ref-16)
17. Директива switch отсутствует в Таблице 5-8 в англоязычной версии руководства. (Прим. перев.) [↑](#footnote-ref-17)
18. В более ранних версиях HI-TECH C компоновщик назывался LINK.EXE [↑](#footnote-ref-18)