



AMIGA:

Программирование на ассемблере

БИБЛИОТЕКА ЖУРНАЛА

AMIGA
Guide

AMIGA:

Программирование на ассемблере

БИБЛИОТЕКА ЖУРНАЛА



СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ РЕДАКЦИИ.	5
РАЗДЕЛ 1. ВВЕДЕНИЕ.	7
1.1. Что такое язык ассемблера и для чего он нужен? ...	7
1.2. Память Amiga.	8
1.2.1. RAM, ROM, внешние регистры.	8
1.2.2. Биты, байты и слова.	10
1.2.3. Системы счисления.	11
1.3. Amiga изнутри.	13
1.3.1. Компоненты и библиотеки.	13
1.3.2. Память.	14
1.3.3. Многозадачность.	16
РАЗДЕЛ 2. ПРОЦЕССОР MC68000.	19
2.1. Регистры.	19
2.2. Адресация памяти.	22
2.3. Режимы процессора.	31
2.3.1. Режимы пользователя и супервизора.	31
2.3.2. Исключения (exceptions)	32
2.3.3. Прерывания.	36
2.3.4. Коды условий.	37
2.4. Команды процессора MC680x0.	40
РАЗДЕЛ 3. СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ.	59
3.1. Ассемблер ASSEM.	59
3.2. Система AssemPro.	61
3.3. Система K-SEKA.	64
РАЗДЕЛ 4. ПЕРВЫЕ ПРОГРАММЫ.	73
4.1. Суммирование таблиц (массивов).	73
4.2. Сортировка таблиц (массивов).	76
4.3. Перевод систем счисления.	79
4.3.1. Перевод шестнадцатеричных чисел в ASCII-представление.	80
4.3.2. Перевод десятичных чисел в ASCII-представление.	83
4.3.3. Перевод ASCII-строк в шестнадцатеричные числа.	85
4.3.4. Преобразование ASCII-строк в десятичные числа.	89

РАЗДЕЛ 5. ВНЕШНИЕ РЕГИСТРЫ.	91
5.1. Работа со специальными клавишами.	91
5.2. Работа с таймером.	92
5.3. Работа с мышью и джойстиком.	94
5.4. Работа со звуком.	97
5.5. Обзор внешних регистров.	104
РАЗДЕЛ 6. ОПЕРАЦИОННАЯ СИСТЕМА.	108
6.1. Загрузка библиотек.	108
6.2. Вызов библиотечных функций.	111
6.3. Инициализация программ.	113
6.3.1. Резервирование памяти.	113
6.3.2. Создание простого окна ввода/вывода.	115
6.4. Ввод/вывод.	120
6.4.1. Вывод текста на экран.	121
6.4.2. Ввод с клавиатуры.	132
6.4.3. Работа с принтером.	138
6.4.4. Работа с последовательным портом.	138
6.4.5. Синтез речи.	138
6.5. Работа с дисководом.	150
6.5.1. Открытие и закрытие флэйвов.	151
6.5.2. Чтение и запись данных.	152
6.5.3. Удаление флэйвов.	154
6.5.4. Переименование флэйвов.	155
6.5.5. Команды CLI.	155
6.5.6. Чтение каталогов диска.	158
6.5.7. Непосредственный доступ к диску.	166
РАЗДЕЛ 7. РАБОТА с INTUITION.	175
7.1. Работа с экранами (screens).	176
7.2. Создание окон.	185
7.3. Работа с реквестерами (requesters).	190
7.4. Обработка событий.	193
7.5. Работа с меню.	195
7.6. Вывод текста.	211
7.7. Работа с графикой.	212
7.8. Работа с бордюрами.	215
7.9. Gadget'ы.	218
7.9.1. Булевские gadget'ы.	218
7.9.2. Текстовые gadget'ы.	225
7.9.3. Пропорциональные gadget'ы.	230
7.10. Пример программы.	233

РАЗДЕЛ 8. ДОПОЛНЕНИЯ.	245
8.1. Режим супервизора.	245
8.2. Программирование исключений.	246
РАЗДЕЛ 9. AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ? ...	250
Процессор.	251
Память.	256
Звук.	257
Видео.	259
Несколько слов о многозадачности.	261
Что делать?	262
ПРИЛОЖЕНИЯ.	265
1. Обзор библиотечных функций AMIGA OS 1.3.	265
2. Обзор команд процессора MC68000.	278
3. Обзор команд всего ряда 680x0.	281
4. Сводка команд, появившихся в процессоре 68020 и операции с сопроцессором.	286
5. Несколько слов о процессоре 68060.	287
6. Расчёт времени выполнения инструкций процессора 68000.	289
ВРЕМЯ ВЫПОЛНЕНИЯ ИНСТРУКЦИЙ ПЕРЕСЫЛКИ. ..	290
ВРЕМЯ ВЫПОЛНЕНИЯ ОБЫЧНЫХ ИНСТРУКЦИЙ.	291
ВРЕМЯ ВЫПОЛНЕНИЯ НЕПОСРЕДСТВЕННЫХ ИНСТРУКЦИЙ.	293
ВРЕМЯ ВЫПОЛНЕНИЯ ОДНООПЕРАНДНЫХ ИНСТРУКЦИЙ.	294
ВРЕМЯ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ СДВИГА.	295
ВРЕМЯ ВЫПОЛНЕНИЯ БИТОВЫХ ОПЕРАЦИЙ.	295
ВРЕМЯ ВЫПОЛНЕНИЯ УСЛОВНЫХ ИНСТРУКЦИЙ.	296
ВРЕМЯ ВЫПОЛНЕНИЯ ИНСТРУКЦИЙ JMP, JSR, LEA, PEA И MOVEM.	297
ВРЕМЯ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ С ПОВЫШЕННОЙ ТОЧНОСТЬЮ.	298
ВРЕМЯ ВЫПОЛНЕНИЯ ДРУГИХ РАЗЛИЧНЫХ ИНСТРУКЦИЙ.	298
ВРЕМЯ ВЫПОЛНЕНИЯ ИНСТРУКЦИЙ ОБМЕНА С ПЕРИФЕРИЕЙ.	300
ВРЕМЯ ОБРАБОТКИ ПРЕРЫВАН.	300

Предисловие редакции.

Ассемблер — это “родной” язык Amiga, дающий программисту возможность в полной мере использовать ресурсы и скорость компьютера.

Программирование на ассемблере требует знаний процессора MC68000 и внутренних характеристик Amiga. Большое число функций, предлагаемых операционной системой (ОС), доступны и широко используются при программировании на ассемблере.

Стандартная документация по функциям ОС написана для использования в С-программах, и практически бесполезна при программировании на ассемблере. В этой книге мы рассмотрим процессор MC68000, операционную систему и ее функции с точки зрения программиста на ассемблере.

Для начала мы рассмотрим организацию памяти и основные функции компьютера, затем обратимся более детально к внутренней структуре Amiga и ее процессора. Описание сопровождается множеством примеров, которые, несомненно, помогут Вам лучше понять материал.

После обзора команд процессора мы покажем, как работать с окнами и меню. Приведенные примеры позволяют создать библиотеку процедур, которую можно впоследствии использовать для написания быстрых, профессиональных программ с дружественным графическим интерфейсом.

Мы надеемся, что эта книга поможет многим пользователям Amiga в изучении машинного программирования

AMIGA: ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ.

и в создании новых полезных программ.

При написании данной книги использовались материалы и работы как отечественных, так и зарубежных программистов и поэтому она имеет коллективное авторство. В частности, использованы главы из работы Стефана Диттера, которая признана одной из лучших для изучения языка ассемблера компьютеров Amiga. В ней в наиболее понятной форме приведены как основные понятия машинного программирования, так и достаточно подробное руководство по написанию ассемблерных программ, корректно работающих под управлением операционной системы AmigaDOS.

Несмотря на то, что материалы данной книги содержат информацию преимущественно о младших моделях Amiga (A500, A500+, A600), она несомненно окажется полезной и для пользователей "настоящих" Амиг (A1200, A4000, ...). Дело в том, что основы машинного программирования одинаковы для всех конфигураций Amiga, а использование запросов старых систем (1.2 и 1.3), которые подробно описываются в этой книге, гарантирует корректную работу Ваших программ на всех моделях Amiga.

В связи с тем, что в последнее время наибольшее распространение получили Amiga на базе процессоров MC68020 и выше, в книгу введены все возможные замечания, касающиеся этого процессора (а следовательно и 68030, 68040 итд.).

ВВЕДЕНИЕ**Раздел I. ВВЕДЕНИЕ.**

Прежде чем перейти непосредственно к описанию ассемблера, отметим ряд вещей, которые составляют основу машинного программирования. Этот раздел мы ввели исключительно для полноты изложенного материала. Подготовленный читатель может его с легким сердцем пропустить.

1.1. Что такое язык ассемблера и для чего он нужен?

Единственным языком, "понятным" процессору Amiga, является так называемый машинный язык. Программы, написанные на других языках (BASIC, Pascal, C), должны быть переведены на машинный язык. Такой перевод может осуществляться либо во время выполнения программы (например, интерпретатором языка BASIC), либо до ее выполнения (компиляторами C и Pascal).

Любая программа на машинном языке — это последовательность нулей и единиц, понятная процессору. Но писать программы в таком виде, согласитесь, весьма нелегко и непривычно. Поэтому почти все машинные программы пишутся на некоем символьном языке, в котором для каждой команды существует своя мнемоника (например MOVE, ADD, CMP, итд.). Такой язык называют языком ассемблера, а для перевода текстов программ в машинный код используются специальные программы — ассемблеры (assembler — сборщик).

Преимущества.

Главным преимуществом программ, написанных на ассемблере, является скорость их работы. В случае интерпретатора языка BASIC, для перевода каждой строки про-

ВВЕДЕНИЕ

граммы требуется время, а компиляторы C и Pascal генерируют громоздкий и неоптимальный код, так что в результате оказывается, что программы работают медленнее, чем аналогичные, но написанные целиком на ассемблере.

Еще одним преимуществом ассемблера над BASIC'ом является то, что для выполнения машинных программ не требуется интерпретатор.

Программы, написанные на ассемблере, имеют доступ ко всем ресурсам компьютера. При этом можно писать подпрограммы на ассемблере для дальнейшего их использования в языках высокого уровня.

1.2 Память Amiga.

Перед написанием любой машинной программы Вам необходимо знать, что от нее требуется, а также необходимо определить ресурсы, которые будут использоваться программой. Одним из самых важных ресурсов является память компьютера.

1.2.1. RAM, ROM, внешние регистры.

RAM (ОЗУ).

Оперативная память (RAM — Random Access Memory) используется для хранения информации с возможностью ее изменения. При этом информация в RAM сохраняется до выключения питания компьютера.

При включении питания Amiga выполняет ряд действий (инициализация, обращение к диску итд.). Для хранения программ, которые выполняются до загрузки операционной системы, очевидно, нужна память, сохраняющая данные и при выключенном питании. Такая память называется постоянной (ROM).

ROM (ПЗУ).

Как видно из аббревиатуры (ROM — Read Only Memory — память только для чтения), данные из ROM можно читать, но не записывать или изменять. Информация в ROM заносится при изготовлении микросхем памяти (или при программировании с помощью специальных уст-

ВВЕДЕНИЕ

ройств). Внутри Amiga имеется микросхема ROM, в которой записана программа начальной загрузки операционной системы (Workbench) и базовой системы Kickstart (ранние модели Amiga не содержали Kickstart в ROM).

PROM.

Одной из разновидностей ROM-памяти является программируемая постоянная память (PROM). Данные заносятся в PROM-микросхемы один раз без возможности стирания, в связи с чем этот тип постоянной памяти используется не очень часто. Вместо PROM памяти чаще используют EPROM память (Erasable Programmable ROM — программируемые ROM с возможностью стирания). Обычно встречаются EPROM-микросхемы с ультрафиолетовым стиранием.

EEPROM.

Значительно менее доступными и более дорогими являются микросхемы EEPROM-памяти — памяти с электрическим стиранием (Electrically Erasable ROM). Такая память работает как оперативная (RAM) с той разницей, что информация сохраняется и при отсутствии питания.

WOM.

С появлением компьютеров Amiga был разработан еще один тип памяти — WOM (Write Once Memory). Информация в такую память загружается с диска при включении компьютера, и после этого никакие данные не могут быть туда записаны. На самом деле, WOM не является новой технологией: WOM-память — это обыкновенная оперативная память с блокировкой записи после первой загрузки.

Внешние регистры.

В дополнении к RAM- и ROM-памяти существует еще один, промежуточный тип памяти — внешние регистры. Используя внешние регистры можно программировать аппаратную часть (hardware) компьютера. Позже мы рассмотрим программирование hardware более подробно.

Теперь обратимся к структуре и использованию оперативной памяти Amiga.

ВВЕДЕНИЕ

1.2.2 Биты, байты и слова.

Килобайт.

Стандартная единица измерения объемов памяти — килобайт. Один килобайт содержит 1024 байт, а не 1000, как может показаться. Такая необычная система измерений связана с двоичным представлением информации внутри компьютера. Для доступа к блоку памяти размером в 1 килобайт, процессору нужно 10 разрядов шины, так что $2^{10}=1024$ байт.

Байт.

Байт, в свою очередь, состоит из восьми полей, принимающих значения 0 или 1 (8 бит). Таким образом байт может принимать $2^8=256$ различных значений, представляющих числа от 0 до 255.

Память Amiga размером 512 килобайт содержит $2^{19}=524288$ байт и 4194304 бит. В памяти такого размера может храниться $2^{24}=16777216$ различных комбинаций нулей и единиц.

Слово.

Знания битов и байтов достаточно для программирования восьмиразрядных процессоров, таких, как 6500. Но для программирования 16/32 разрядного процессора MC68000 Вам понадобятся еще две формы представления данных: слово (2 байта, или 16 бит) и длинное слово (4 байта, или 32 бита).

Слова могут принимать значения в диапазоне от 0 до 65535, а длинные слова — до 4294967295. Процессор MC68000 может обрабатывать такие гигантские числа одной операцией.

Нередко возникает потребность в использовании отрицательных чисел наряду с положительными. Поскольку биты не могут принимать отрицательных значений, используется специальная схема представления чисел, в которой самый старший, 15-й (от нуля) бит используется как знаковый (0 — число положительное, 1 — отрицательное). При этом отрицательные числа задаются как бы задом-наперед:

ВВЕДЕНИЕ

число -1 представляется как \$FFFF, -2 — как \$FFFE, итд. до значения -65536, представляемого словом \$8000. Такое странное, на первый взгляд, представление используется для облегчения аппаратной реализации команд (не требуются специальных алгоритмов для обработки отрицательных чисел).

В языке ассемблера редко используется привычная десятичная система счисления, вместо нее используются двоичная, восьмеричная и шестнадцатеричная системы.

1.2.3 Системы счисления.

Рассмотрим десятичную систему счисления, в которой основным (базовым) числом является 10. Это означает, что каждый разряд задает степень числа 10 (например, 246 задает число $2 \cdot 10^2 + 4 \cdot 10^1 + 6 \cdot 10^0$). Десятичная система использует таким образом 10 цифр (от 0 до 9) для записи разрядов.

Двоичная система.

Двоичная система счисления использует только две цифры для записи разрядов: 0 и 1. Таким образом базой двоичной системы является число 2. Например, десятичная запись двоичного числа 1010 определяется так:

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 2^3 + 2^1 = 8 + 2 = 10$$

Далее мы будем записывать двоичные числа с символом % в начале, например %110010. Для перевода числа из двоичной системы в десятичную нужно просто сложить значения разрядов, содержащих 1. Значения разрядов можно получить из следующей таблицы:

номер разряда	7	6	5	4	3	2	1	0
значение	128	64	32	16	8	4	2	1

Для тренировки, попробуйте перевести число %110010 в десятичную систему счисления (у Вас должно получиться число 50).

Восьмеричная система.

Базой восьмеричной системы счисления является число 8. Разряды числа в восьмеричной записи могут содержать цифры от 0 до 7. Таким образом, десятичным эк-

ВВЕДЕНИЕ

ивалентом восьмеричного числа 31 является $3 \cdot 8^1 + 1 \cdot 8^0 = 25$.

Шестнадцатеричная система.

Базовое число шестнадцатеричной системы — 16, а возможные цифры (символы) в разрядах — от 0 до F. Так, символ A эквивалентен десятичному числу 10, а символ F — числу 15. Шестнадцатеричные числа мы будем писать со знаком \$ в начале. Двоичная и шестнадцатеричная системы счисления являются базовыми в машинном программировании.

Шестнадцатеричное представление байта всегда содержит два разряда (две позиции), диапазон 0-255 представляется диапазоном \$00-\$FF. Слово представляется диапазоном \$0000-\$FFFF, а двойное слово — \$00000000-\$FFFFFFFF.

Перевод двоичные чисел в шестнадцатеричные осуществляется по очень простой схеме: исходное число разбивается на группы по четыре разряда. Каждой из этих групп соответствует шестнадцатеричная цифра. Например:

двоичное число	%110011101111
разбиение	%1100 %1110 %1111
результат	\$C \$E \$F

таким образом %110011101111 = \$CEF

Обратный перевод осуществляется аналогично:

шестнадцатеричное	\$E30D
разбиение	\$E \$3 \$0 \$D
результат	%1110 %0011 %0000 %1101

таким образом \$E30D = %1110001100001101

Аналогичный метод используется для перевода двоичных чисел в восьмеричную систему и наоборот, только в этом случае группы разбиения содержат по 3 разряда:

восьмеричное число	7531
разбиение	7 5 3 1
результат	%111 %101 %011 %001

таким образом восьмеричное 7531 = %111101011001

Полученное двоичное число можно перевести в шестнадцатеричную систему:

ВВЕДЕНИЕ

двоичное число	%111101011001
разбиение	%1111 %0101 %1001
результат	\$F \$5 \$9

таким образом восьмеричное $7531 = \$F59$

Затем можно перевести полученное число в привычную десятичную систему по следующей схеме:

шестнадцатеричное	\$F59
разбиение	\$F \$5 \$9
результат	$15 \cdot 16^2 + 5 \cdot 16 + 9$

таким образом $\$F59 =$ десятичному 3929

Несмотря на простоту этих преобразований, каждый раз проводить их вручную весьма утомительно. Для этого обычно используются встроенные средства (например, некоторые ассемблеры при вводе числа с символом '?' выдают его в десятичном и шестнадцатеричном виде).

Зачастую подобные преобразования используются внутри программ, например, когда Вам нужно ввести число с клавиатуры для дальнейшей обработки программой. В этом случае число вводится в виде последовательности символов и затем преобразуется в двоичную форму.

Подобное происходит и при выводе чисел, только в обратном порядке: Вам сначала нужно преобразовать выводимое число в последовательность символов, а затем вызвать подпрограмму печати. В следующих разделах мы подробно остановимся на написании машинных программ, выполняющих такие преобразования.

1.3 Amiga изнутри.

Чтобы программировать на ассемблере, недостаточно знать только лишь команды центрального процессора. Необходимы знания и о других компонентах компьютера.

1.3.1 Компоненты и библиотеки.

Высокая производительность Amiga достигается благодаря устройствам, берущим на себя определенную работу, и освобождающим от нее центральный процессор. Такие устройства называются custom-чипами (custom chips).

ВВЕДЕНИЕ

Custom-чипы.

Процессору MC68000 помогают три микросхемы, названные создателями Amiga как Agnus, Denise и Paula. Главной задачей чипа Agnus является пересылка блоков памяти, Denise отвечает за вывод данных на экран, а Paula — за внешний ввод/вывод (дисковые операции, звук, итд.). Эти чипы доступны процессору через hardware-регистры с адресами от \$DFF000 (более детально hardware-регистры описаны в соответствующем разделе). Для облегчения работы с custom-чипами в состав Workbench- и Kickstart-библиотек входят специальные функции.

За последнее десятилетие количество, названия и функции custom-чипов подверглись существенным изменениям, но совместимость “снизу вверх” по-прежнему сохраняется. Подробнее смотрите раздел 9.

Эти библиотечные функции написаны на ассемблере и легко сопрягаются с пользовательскими программами. Конечно, можно обойтись и без библиотечных функций, написав собственные, однако программирование custom-чипов напрямую — задача весьма трудоемкая.

Подробнее о правомерности программирования custom-чипов напрямую Вы можете прочитав в разделе 9.

1.3.2. Память.

Для начала рассмотрим память Amiga A1000. Стандартная конфигурация этого компьютера содержит 512 килобайт ЧАМ, расположенные по адресам \$00000- \$7FFFF, или 0-524287. Если память расширена до 1 мегабайта, первые 512 килобайт расположены там же, однако вторая половина памяти может располагаться с любого адреса между \$200000 и \$9FFFFFF. В системах AmigaDOS, начиная с версии 1.2, используется специальная схема автоконфигурации, что позволяет расширять память, не заботясь об адресах.

Chip-память.

Custom-чипы обращаются к памяти независимо от процессора, что позволяет им работать практически одновременно. Однако custom-чипы могут адресовать только

ВВЕДЕНИЕ

первые 512 килобайт памяти. Таким образом, графика и звуковые данные должны размещаться в этой области памяти, которая получила название chip RAM.

Более новые модификации Amiga — A500+, A600 используют ECS (Enhanced Chipset) вместо OCS (Original Chipset). ECS-чипы позволяют адресовать до двух мегабайт chip-памяти. На A1200 и A4000, оснащенных AGA-чипсетом, уже штатно стоит 2 мегабайта chip-памяти.

Fast-память.

Память, к которой имеет доступ только центральный процессор, называется fast RAM. Обычно fast RAM располагается по адресу \$200000.

Размещение Fast RAM с адреса \$200000 позволяет установить только до восьми мегабайт памяти. На более современных системах с процессорами 68020 и выше Fast RAM может располагаться за пределами 16-мегабайтного адресного пространства, и объем ее практически неограничен.

Рассмотрим общую карту памяти Amiga:

\$000000-\$07FFFF	chip-память
\$080000-\$1FFFFFF	зарезервировано
\$200000-\$9FFFFFF	место для fast-памяти
\$A00000-\$BEFFFF	зарезервировано
\$BFD000-\$BFDFFF	PIA В (четные адреса)
\$BFE001-\$BFEFFF	PIA С (нечетные адреса)
\$C00000-\$DEFFFF	зарезервировано для расширений
\$DFF000-\$DFFFFF	hardware-регистры
\$E00000-\$EFFFFF	зарезервировано
\$E80000-\$EFFFFF	порты расширений
\$F00000-\$F7FFFF	зарезервировано
\$F80000-\$FFFFFF	системное ПЗУ (ROM)

Операционная система Amiga поддерживает многозадачность, так что место размещения программ в оперативной памяти не фиксировано. Блок памяти, занимаемый кодом программы, помечается как занятый в специальном списке и становится недоступным для других программ:

ВВЕДЕНИЕ

при загрузке программы система, анализируя этот список, выбирает свободную область памяти нужного размера. Если программе в процессе работы требуется дополнительная память (например, для хранения текстового буфера), необходимо выполнить специальный запрос системы — резервирование. В противном случае не гарантируется, что данная область памяти не будет использоваться другими программами.

Когда программа заканчивает работу, занимаемая ей память освобождается для дальнейшего использования. В результате вся память оказывается разбитой на свободные и занятые блоки, не связанные друг с другом (фрагментация). Правда, если появляются несколько расположенных друг за другом свободных блоков, система объединяет их в один. Примером работы данного механизма может служить динамический RAM-диск.

Особенностью RAM-диска является то, что он всегда заполнен. Когда программа удаляется с RAM-диска, память, отведенная для ее хранения, возвращается системе. При записи данных на RAM-диск память, наоборот, резервируется у системы.

В младших моделях Amiga нет механизма страничной организации памяти: при резервировании блока системы пытается выделить непрерывный кусок памяти требуемого размера. Так что возможна ситуация, когда суммарный объем свободной памяти позволяет выделить блок нужного размера, но каждый из свободных блоков памяти оказывается меньше, и запрос возвращает ошибку.

1.3.3 Многозадачность.

Многозадачность (возможность выполнять несколько программ одновременно) является основой операционной системы Amiga. Именно операционная система занимается распределением ресурсов компьютера между программами. Центральный процессор MC68000 не может выполнять несколько программ одновременно, но несмотря на это, при наличии некоторых аппаратных средств, можно “заста-

ВВЕДЕНИЕ

вить" процессор время от времени переключаться с одной программы на другую. Многозадачность на Amiga реализована именно так: когда работают несколько задач, операционная система распределяет между ними процессорное время. Рассмотрим пример: пусть в одном окне работает текстовый редактор, а в другом — программа копирования дискет. Допустим, что процессор выполняет чтение порции данных с диска (работает вторая программа). Как только чтение завершается, управление передается текстовому редактору, который проверяет состояние клавиатуры и обрабатывает вводимые команды. Затем управление снова передается копировщику для чтения очередной порции данных, и так далее. На самом деле переключение задач осуществляется очень часто (порядка 50 раз в секунду), что создает впечатление одновременной работы программ.

Задачи.

Программы, работающие под управлением операционной системы, называют задачами (tasks). При одновременной работе нескольких задач, каждой из них выделяется определенный квант времени, в течении которого она выполняется. Длительность этого кванта регулируется системой, так что при необходимости более важным задачам может выделяться больше процессорного времени.

На самом деле, программисту вовсе необязательно знать детали распределения времени. Можно писать программы вообще забыв про многозадачность, ведь распределением ресурсов занимается операционная система. Единственным ограничением является то, что программа должна быть запущена либо из CLI (Command Line Interpreter — интерпретатор командной строки) командой "run", либо из Workbench.

Существует еще одна важная деталь: пользовательские программы не должны программировать hardware напрямую, вместо этого следует использовать библиотечные функции. Причина этого проста.

Пусть, например, одна задача занимается чтением

ВВЕДЕНИЕ

данных из принтерного порта, а другая — печатью на принтер. Если хотя-бы одна из этих программ работает с портом напрямую, то неизбежно возникнет путаница.

Данный пример очень упрощен, на практике последствия подобной несогласованной работы бывают куда более катастрофическими. Если все же Вам требуется работать с hardware напрямую (что дает некоторые преимущества), то для начала убедитесь, что не возникнет подобных коллизий с другими задачами.

ПРОЦЕССОР MC68000

Раздел 2. ПРОЦЕССОР MC68000.

MC68000 — это 16/32 разрядный процессор. Это означает, что имея возможность обрабатывать 32-битные данные, MC68000 содержит 16-битную шину данных и 24-битную шину адреса. Таким образом, MC68000 может непосредственно адресовать $2^{24} = 16777216$ байт (16 мегабайт) памяти.

7.1 мегагерц.

Тактовая частота, на которой работает процессор Amiga, составляет всего 7.1 мегагерц. Но несмотря на это, общая производительность Amiga достаточно высока: custom-чипы почти полностью освобождают процессор от работы с графикой, анимацией и звуком.

Начиная с 68020, процессоры этой серии адресуют 4 гигабайта, а тактовая частота может достигать 80 мегагерц (для 68060).

2.1 Регистры.

В дополнении к оперативной памяти, внутри процессора существует еще и регистровая память. Процессор MC68000 содержит восемь регистров данных (D0-D7), восемь регистров адреса (A0-A7), регистр статуса (SR), а также программный счетчик (PC). Регистр A7 играет особую роль — роль указателя стека.

Размеры регистров.

Все регистры, за исключением SR, 32-битные. Адресуются регистры специальным образом, так как они расположены непосредственно внутри процессора, в то время как обычная оперативная память — это отдельное устройство.

ПРОЦЕССОР MC68000

Регистры данных.

Регистры данных используются для хранения любых данных в виде байтов, слов и длинных слов.

Адресные регистры.

Адресные регистры служат для хранения и обработки адресов (указателей).

Указатель стека.

Адресный регистр A7 используется процессором как указатель стека, и в связи с этим не рекомендуется его применять для других целей. На самом деле, процессор 68000 имеет два стека — пользовательский и системный. То, на какой из этих стеков указывает A7, зависит от текущего режима процессора (об этом речь пойдет в следующем пункте).

Стек — это специальная область памяти, которая используется для хранения внутренних промежуточных данных. Указатель стека содержит адрес вершины стека, то есть адрес последнего занесенного в стек слова. При “заталкивании” новых данных в стек, указатель уменьшается (стек “растет”), а при “выталкивании” из стека наоборот, увеличивается. Можно провести аналогию с кипой бумаг на столе: бумага, которую Вы положили последней, берется из кипы первой. Такой механизм называется LIFO (Last In First Out — последним вошел, первым вышел).

О том, как работать с указателями стека, мы расскажем в следующем разделе.

Программный счетчик.

Регистр программного счетчика (PC) всегда указывает на адрес команды, которая следует за текущей (выполняемой в данный момент) командой.

Регистр статуса.

Регистр статуса (SR) играет особую роль в машинном программировании. SR — это 16-битный регистр, который содержит информацию о состоянии процессора (используются только 10 бит из 16-ти). Отдельные биты регистра состояния называются флагами условий. Если какое-то

ПРОЦЕССОР MC68000

условие истинно, соответствующий флаг содержит 1.

В следующей таблице приведено описание битов регистра SR:

Бит	Название	Смысл
0	C, Carry (перенос)	Флаг переноса, устанавливается арифметическими операциями и командами сдвига.
1	V, Overflow (переполнение)	Флаг арифметического переполнения (или изменения знака).
2	Z, Zero (нуль)	Флаг установлен, если результат последней операции — 0.
3	N, Negative (знак)	Знаковый флаг: устанавливается, если результат последней операции отрицателен.
4	X, Extended (дополнительный)	Устанавливается арифметическими операциями как копия бита C.
5-7		Не используются
8-10	I0,I1,I2	Маска прерываний. Задаёт уровень прерываний (от 0 до 7, причем значение 7 задаёт наивысший приоритет).
11,12		Не используются
13	S, Supervisor (режим)	Режим процессора (0 — пользовательский, 1 — режим супервизора).
14		Не используется
15	T, Trace (трассировка)	Установлен, если процессор работает в режиме трассировки (пошагового выполнения команд).

Регистр статуса можно представить так:

ПРОЦЕССОР MC68000

бит: | 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
название: | T -- S -- -- I2 I1 I0 -- -- -- X N Z V C

В следующих версиях процессоров набор управляющих регистров расширен. Более подробно об этом мы расскажем в разделе 9 и в приложении.

2.2. Агрессия памяти.

В стандартной конфигурации модели Amiga 500 и Amiga 1000 содержат 512 килобайт оперативной памяти, а Amiga 2000 — 1 мегабайт (512 килобайт chip-памяти и столько же fast-памяти). Каким же образом процессор работает с памятью?

При программировании на BASIC'e Вам не нужно заботиться о распределении данных в памяти. Можно просто написать `MARKER%=1` и интерпретатор сам разместит это значение в оперативной памяти.

При программировании на ассемблере существует два способа хранения данных:

- 1) в одном из регистров данных (или адреса);
- 2) непосредственно в оперативной памяти.

Забегая немного вперед, рассмотрим машинную команду `MOVE`, которая, по всей видимости, является наиболее часто используемой в программировании на ассемблере. Команда `MOVE` служит для пересылки (копирования) данных и имеет два аргумента: источник (откуда копировать) и приемник (куда копировать).

При использовании `MOVE` приведенный выше пример из BASIC'a переписывается так:

`MOVE #1,D0`

Команда `MOVE` в такой записи копирует число 1 в регистр `D0`. Операнд источника всегда указывается раньше операнда приемника. Это означает, что нельзя писать так:

`MOVE D0,#1.`

Команда `MOVE #1,$1000` записывает число 1 в ячейку памяти с адресом `$1000`. Адрес в данном примере выбран произвольно, и обычно в такой форме в реальных программах не записывается. Вместо этого используют метки:

ПРОЦЕССОР MC68000

```
...
MOVE #1,MARKER
```

```
...
MARKER: DC.W 1
```

Программа состоит, в данном случае, из двух частей: первая часть — это обычная команда MOVE, которая копирует число 1 в ячейку, помеченную как MARKER. Вторая часть содержит псевдооперацию (pseudo-op) DC.W, которая не является командой процессора. Псевдооперации — это директивы ассемблера (напомним, что ассемблер — это специальная программа-компилятор для перевода машинных команд из символьного представления в двоичное). В нашем случае DC означает DeClare (определить переменную), а суффикс .W задает размер переменной — одно слово (word). Возможны другие варианты суффиксов: .B (байт) и .L (длинное слово).

Такие же суффиксы (.B .W или .L) используются и при записи большинства машинных команд для задания размера операции, например

```
MOVE.L #$12345678,D0
```

пересылает длинное слово, а

```
MOVE.B #$12,D0
```

— байт. Если суффикс опущен, ассемблер использует размер по умолчанию — .W.

Предостережение.
Если при работе с памятью используется размер .W (или .L), адрес должен быть четным (оканчиваться на 0,2,4,6,8,A,C,E)!

Ассемблеры обычно имеют директиву 'EVEN' (или 'ALIGN'), которая служит для выравнивания адресов по четности. Такая директива необходима, например, в следующей ситуации:

```
...
VALUE1: DC.B 1
VALUE2: DC.W 1
```

Если переменная VALUE1 размещена по четному адресу, то VALUE2 автоматически попадает на нечетный адрес. Если же использовать ALIGN (EVEN), то между VALUE1 и VALUE2 (в случае необходимости) будет "встав-

ПРОЦЕССОР MC68000

лен" нулевой байт, и адрес VALUE2 станет четным:

```
...  
VALUE1: DC.B 1  
ALIGN  
VALUE2: DC.W 1
```

При выборке отдельных байтов (слов) из памяти необходимо иметь в виду порядок их расположения. В процессорах MC680x0 принят такой порядок, при котором в младших адресах содержится старшая часть слова (длинного слова). Это означает, например, что после выполнения команды `MOVE.W #$1234,$1000` по адресу \$1000 будет размещен байт \$12, а по адресу \$1001 — байт \$34. То же самое относится и к "длинным" пересылкам: например, память после выполнения `MOVE.L #$12345678,$1000` будет выглядеть так:

```
$1000: $12  
$1001: $34  
$1002: $56  
$1003: $78
```

Вернемся к способам адресации. Все приведенные примеры эквивалентны оператору BASIC'a `MARKER%=1`, где символ % показывает, что переменная `MARKER` — целочисленная.

Сделаем еще один шаг и переведем оператор `MARKER%=VALUE%` на ассемблер. Результат очевиден: `MOVE VALUE, MARKER`

```
...  
...  
MARKER:   DC.W 1  
VALUE:    DC.W 1
```

В данном случае содержимое переменной `VALUE` копируется в переменную `MARKER`.

С помощью этих примеров Вы уже ознакомились с некоторыми методами адресации (доступа) к памяти. Первый метод использует, так называемый, непосредственный операнд (со знаком #), и называется прямым (непосредст-

ПРОЦЕССОР MC68000

венным) методом адресации. Как уже отмечалось, непосредственным может быть только операнд источника.

Второй рассмотренный метод — это метод абсолютной адресации, который использует для записи операндов их абсолютные адреса (в нашем примере это MARKER и VALUE). Абсолютная адресация может использоваться как для операндов источника, так и для операндов приемника.

На самом деле существуют две формы абсолютной адресации, разница между которыми обычно не играет роли для программиста. В зависимости от значения абсолютного адреса используются длинная или короткая форма абсолютной адресации (если адрес меньше, чем \$FFFF, то используется короткая форма). Ассемблер сам выбирает нужную форму адресации.

Четвертый метод адресации, который мы использовали, называется регистровым (команда MOVE #1,D0 использует этот метод для задания регистра D0 в качестве операнда приемника).

Существует еще одна разновидность регистрового метода адресации, которая используется для доступа к адресным регистрам: команда MOVE.L #MARKER,A0 записывает адрес переменной MARKER в регистр A0.

Теперь переходим к менее тривиальным (и более интересным) методам адресации. Рассмотрим еще один пример из BASIC'a:

```
10 A=1000  
20 POKE A,1
```

В первой строчке переменной A присваивается значение 1000. Такое присвоение возможно и на ассемблере: MOVE.L #1000,A0 (здесь переменная A размещается в адресном регистре A0).

В строчке 20 значение 1 записывается не в саму переменную A, а в ячейку памяти с адресом, содержащимся в A. Это еще один способ доступа к данным, который легко записывается и на ассемблере:

```
MOVE.L #1000,A0 ;записать адрес в A0
```

ПРОЦЕССОР MC68000

MOVE #1,(A0) ;записать 1 по этому адресу

Скобки задают косвенный регистровый метод адресации. Этот метод работает только с адресными регистрами.

Существует несколько разновидностей косвенных регистровых методов. Например, может быть задано смещение, которое добавляется к адресному регистру перед его использованием. Так, команда MOVE #1,4(A0) запишет значение 1 по адресу $1000+4=1004$. Смещение может быть как положительным, так и отрицательным и должно лежать в диапазоне -32768..32767. Этот метод адресации называется косвенным регистровым с 16-битным смещением.

Существует еще один похожий метод адресации — косвенный регистровый с 8-битным смещением. В этом методе в качестве дополнительного смещения может использоваться значение какого-либо регистра, а размер основного смещения ограничен восемью битами.

Поясним это на примере. Пусть наша программа использует следующую структуру данных (массив):

```
...  
RECORD: DC.B 2 ;число элементов массива  
          ;минус один  
          DC.B 1,2,3 ;элементы массива
```

Будем использовать MOVE.L #RECORD,A0 для загрузки адреса структуры в A0 и MOVE (A0),D0 для получения числа элементов массива. Тогда доступ к последнему элементу может осуществляться одной командой, как показано в следующем примере:

```
...  
CLR.L D0 ;очистка D0  
MOVE.L #RECORD,A0 ;адрес массива — в A0  
MOVE.B (A0),D0 ;число элементов-1 — в D0  
MOVE.B 1(A0,D0),D1 ;последний элемент — в D1
```

...
Здесь последняя команда извлекает элемент, распо-

ПРОЦЕССОР MC68000

ложенный по адресу $1+A0+D0$, или, другими словами, по адресу начала структуры плюс один (здесь начинается массив) и плюс смещение, заданное в $D0$ (в нашем случае 2). Этот метод доступа к данным используется очень часто, он прекрасно подходит для работы с таблицами и списками. Если смещение не требуется, можно написать так: `MOVE 0(A0,D0),D1` (а некоторые ассемблеры подставляют нулевое смещение по умолчанию, так что его можно явно не указывать).

Последние два метода имеют еще одну разновидность, в которой используется смещение относительно содержимого программного счетчика (PC). Используя косвенный регистровый метод со смещением относительно PC можно писать перемещаемые программы. Следующие две команды эквивалентны (при условии, что смещение от адреса второй команды до адреса `MARKER` может быть закодировано 16-ю битами):

`MOVE MARKER,D1`

и

`MOVE MARKER(PC),D1`

Если в первой команде задается абсолютный адрес переменной `MARKER`, то во второй — смещение от самой команды до метки `MARKER` (это смещение вычисляется ассемблером и автоматически подставляется в код).

Рассмотрим подробнее разницу между этими двумя командами. Несмотря на то, что действия их эквивалентны, машинное представление у них разное. Пусть команда `MOVE` расположена с адреса `$1000`, а перемечная `MARKER` — с адреса `$1100`. Тогда машинный код, сгенерированный ассемблером, будет выглядеть примерно так:

`$001000 30 39 00 00 11 00 MOVE MARKER,D1`

и второй вариант:

`$001000 30 3A 00 FE MOVE MARKER(PC),D1`

Отсюда видно, что второй вариант кода на два байта короче первого. Более того, если код первой программы разместить с адреса `$2000`, то операнд команды `MOVE` все

ПРОЦЕССОР MC68000

равно будет браться из ячейки \$1100, в то время как во втором варианте MOVE будет корректно обращаться к переменной MARKER, адрес которой будет теперь уже не \$1100, а \$2100. Таким образом, второй вариант программы будет корректно работать в любом месте памяти.

Существует также вариант косвенного регистрового метода адресации с 8-битным смещением и индексным регистром, в котором все смещения указаны относительно программного счетчика (PC) (например, MOVE M(PC,D0),D1).

Нам осталось рассмотреть еще два метода адресации процессора MC68000: косвенный регистровый с постинкрементом и косвенный регистровый с предекрементом. В первом методе после обращения к памяти адресный регистр увеличивается на 1, 2 или 4 (это зависит от размера команды), например:

```
MOVE.L    #$1000,A0
MOVE.B    #1,(A0)+
```

Здесь по адресу \$1000 запишется 1, а содержимое регистра A0 увеличится на 1 (так как указан суффикс .B). Постинкрементный метод очень полезен при работе с таблицами, когда требуется последовательный доступ к элементам (обычно используется в циклах, но об этом позднее).

Аналогично работает предекрементный метод, в котором перед обращением к памяти адресный регистр уменьшается на 1, 2 или 4, например программа

```
MOVE.L    #$1000,A0
MOVE.B    #1,-(A0)
```

запишет 1 по адресу \$FFF, так как содержимое регистра A0 СНАЧАЛА декрементируется, а затем используется.

Эти методы адресации широко используются для работы со стеком. Например, чтобы "затолкнуть" содержимое D0 в стек, можно написать

```
MOVE D0,-(A7) ;используем предекремент, так
```

ПРОЦЕССОР MC68000

как стек
 ;"растет" в сторону уменьшения
 адресов а чтобы "вытолкнуть"
 слово из стека в D0, можно
 написать
 MOVE (SP)+.D0 ;SP всегда указывает на
 последнее занесенное
 в стек значение

Заметим, что указатель стека должен быть четным (так как стек используется некоторыми командами процессора для размещения слов и длинных слов). Поэтому, если используется постинкрементная/предкрементная адресация стека в байтовых командах, SP будет увеличиваться/уменьшаться на два. Например, команда MOVE.B #1, -(SP) разместит байт 1 в стеке, предварительно уменьшив SP на два (а при использовании любого другого адресного регистра вместо SP размер декремента будет 1).

Как видно из примеров, регистр указателя стека имеет два символических имени: SP и A7. Таким образом, записи MOVE #1, -(SP) и MOVE #1, -(A7) полностью эквивалентны, однако рекомендуется использовать первую форму записи указателя стека (это делает программу более "читабельной", подчеркивая разницу между обычным адресным регистром и SP).

Итак, мы рассмотрели все 12 методов адресации процессора MC68000:

№	Название адресации	Формат
1	регистровая (по регистру данных)	Dn
2	регистровая (по регистру адреса)	An
3	косвенная регистровая	(An)
4	косвенная регистровая с постинкрементом	(An)+
5	косвенная регистровая с предкрементом	-(An)
6	косвенная регистровая с 16-битным	d16(An)

ПРОЦЕССОР MC68000

	смещением и индексированием	
7	косвенная регистровая с 8-битным смещением и индексированием	d8(An,Rn)
8	абсолютная короткая	xxxx.W
9	абсолютная длинная	xxxxxxxx.L
10	непосредственная (прямая)	#'data'
11	косвенная относительно PC с 16-битным смещением и индексированием	d16(PC)
12	косвенная относительно PC с 8-битным смещением и индексированием	d8(PC,Rn)

Условные обозначения:

An - адресный регистр (A0-A7)

Dn - регистр данных (D0-D7)

d16 - 16-битное число

d8 - 8-битное число

Rn - любой регистр (A0-A7, D0-D7)

'data' - число (размер зависит от суффикса команды)

Процессор MC68020, помимо описанных двенадцати, поддерживает еще шесть методов адресации:

— косвенная регистровая с базовым (32-битным) смещением и индексированием (аналогично косвенной регистровой с 8-битным смещением и индексированием);

— косвенная через память с постиндексированием. Например, операнд в команде MOVE ([10,A0,D0,20),D1 берется из ячейки с адресом < содержимое ячейки (10+<содержимое A0>) > + < содержимое D0 > + 20.

— косвенная через память с прединдексированием. Например, команда MOVE ([10,A0,D0],20),D1 берет свой первый операнд из ячейки с адресом < содержимое ячейки (10+<содержимое A0>+<содержимое D0>) > + 20.

— косвенная относительно PC с 32-битным базовым смещением и индексированием;

— косвенная относительно PC через память с по-

ПРОЦЕССОР MC68000

стиндексированием;

— косвенная относительно РС через память с прединдексированием.

Последние три метода аналогичны первым трем, только вместо адресного регистра используется РС.

В методах адресации MC68020, использующих индексирование, рассматривается, так называемый, индексный операнд, который имеет вид $Dn * SCALE$. Здесь параметр SCALE задает масштаб регистра Dn (то есть число, на которое будет умножаться содержимое Dn перед его использованием) и может принимать значения 1, 2, 4 и 8. Например, для доступа к пятому элементу таблицы, содержащей длинные слова, можно использовать команду `MOVE.L (A0,D0*4),D1`, при условии, что $A0$ содержит адрес начала таблицы, а $D0$ — номер элемента (в нашем случае 5).

В командах MC68000 параметр SCALE всегда должен быть равен 1.

Подробную информацию о методах адресации процессора MC68020 можно найти в соответствующей документации.

2.3 Режимы процессора.

В пункте 2.1 мы описывали регистр статуса SR. Теперь подробно остановимся на старшем (системном) байте этого регистра и рассмотрим его влияние на работу процессора.

2.3.1 Режимы пользователя и супервизора.

Вам может показаться странным, что процессор подразделяет программы на обычные — ‘пользовательские’ и специальные — ‘супервизорские’, давая некоторые привилегии командам в режиме супервизора. Однако это не повод для беспокойства — операционная система позволяет в случае необходимости переключаться в режим супервизора.

Режим процессора задается битом 13 регистра статуса. Обычно этот бит содержит 0 (режим пользователя).

ПРОЦЕССОР MC68000

Содержимое регистра SR можно изменять обычными командами (MOVE, AND, итд.), правда такие команды работают только в режиме супервизора (являются привилегированными). Таким образом, можно переключаться из режима супервизора в пользовательский режим командой AND #DFFF,SR, которая очищает 13-й бит регистра статуса. Однако вместо этого рекомендуется использовать функции операционной системы.

Чем же отличаются друг от друга режимы процессора? Только что мы видели, что не все команды процессора доступны в режиме пользователя. Например, реакцией процессора на команду MOVE xx,SR в пользовательском режиме будет генерация исключения (exception) и прерывание программы. В действительности, единственный путь переключения в режим супервизора — через исключения, но об этом речь пойдет позже.

Второе отличие состоит в том, что для каждого из двух режимов используется свой стек. В качестве указателя стека всегда используется регистр A7, значение которого меняется каждый раз при смене режима (в пользовательском режиме A7 указывает на стек пользователя (USP), а в режиме супервизора — на стек супервизора (SSP)).

Текущий режим процессора влияет и на доступ к памяти: при обращении к внешним компонентам (в том числе и к памяти) процессор передает им информацию о текущем режиме. Таким образом можно запретить пользователю доступ к некоторым ячейкам памяти (чисто теоретически, так как в Amiga данный механизм не используется). В режиме супервизора можно выполнять любые команды процессора и адресовать всю память, поэтому операционная система в основном работает в режиме супервизора. Это возможно при использовании механизма исключений.

2.3.2 Исключения (exceptions)

Исключения MC680x0 похожи на прерывания в компьютерах на базе 6500. Механизм исключений позволяет

ПРОЦЕССОР MC68000

прерывать основную программу с вызовом специальной подпрограммы — обработчика исключения. Когда возникает исключение, происходит следующее::

- 1) Сохраняется содержимое регистра статуса;
- 2) Очищаются биты S и T регистра статуса;
- 3) Сохраняется содержимое регистров PC и SP;
- 4) Из таблицы векторов исключений извлекается адрес обработчика;
- 5) Вызывается сам обработчик исключения.

Таблица векторов исключений — это таблица адресов подпрограм-обработчиков для всех возможных исключений. Эта таблица расположена в самом начале оперативной памяти — с адреса 0 (для MC68020 это не совсем так: в MC68020 начальный адрес таблицы исключений берется из специального регистра — VBR). Приведем соответствие векторов и адресов исключений:

№	Адрес	Назначение
0	\$000	RESET: начальный SSP
1	\$004	RESET: начальный PC
2	\$008	ошибка шины
3	\$00C	ошибка адресации
4	\$010	неопознанная команда
5	\$014	деление на ноль
6	\$018	CHK
7	\$01C	TRAPV
8	\$020	нарушение привелегий
9	\$024	трассировка
10	\$028	эмуляция команд Axxx
11	\$02C	эмуляция команд Fxxx
	\$030-\$03B	зарезервировано (не используется)
15	\$03C	прерывание по неинициализации
	\$040-\$05F	зарезервированы
24	\$060	прерывание по верификации
25-31	\$064-\$07F	прерывания уровней 1-7

ПРОЦЕССОР MC68000

32-47	\$080-\$0BF	TRAP
	\$0C0-\$0FF	зарезервировано
64-255	\$100-\$3FF	вектора пользователя

Рассмотрим подробнее каждый элемент таблицы:

RESET: начальный SSP.

При сбросе компьютера запускается специальная программа — RESET. Указатель стека перед выполнением этой программы берется из ячейки с адресом \$000 (начальный SSP).

RESET: начальный PC.

А адрес самой подпрограммы RESET (то есть начальное значение PC) берется из ячейки \$004.

Ошибка шины.

Это исключение генерируется сопроцессором, например, при попытке обращения к несуществующей (зарезервированной) памяти.

Ошибка адресации.

Генерируется при попытке доступа к слову или длинному слову по нечетному адресу.

Неопознанная команда.

Почти все команды процессора MC68000 — 16-битные (без учета операндов), а 16-битное слово может принимать 65536 различных значений. Поскольку реальное число команд процессора намного меньше, то возможна ситуация, когда процессор “натолкнется” на несуществующую команду. В этом случае генерируется исключение “неопознанная команда”.

Деление на ноль.

Генерируется при попытке выполнения команды деления с нулевым делителем.

CHK.

Генерируется командой CHK, если содержимое регистра данных выходит за некоторый диапазон.

TRAPV.

Генерируется по команде TRAPV, если установлен бит V регистра статуса.

ПРОЦЕССОР MC68000

Нарушение привелегий.

Генерируется при попытке выполнения привелегированной команды в пользовательском режиме.

Трассировка.

Если бит трассировки в регистре статуса установлен, то данное исключение генерируется после каждой выполненной машинной команды. Механизм трассировки (пошагового выполнения) используется при отладке программ.

Эмуляция команд Аххх.

Эмуляция команд Fххх.

Если процессор при выполнении программы встречает слово, начинающееся с \$А или \$F (например, \$A010 или \$F200), то генерируется соответствующее исключение. Таким образом, используя вектора \$28 и \$2C, можно расширять набор команд процессора.

Прерывание по неинициализации.

Генерируется, когда неинициализированное внешнее устройство посылает запрос на прерывание.

Прерывание по верификации.

Генерируется при ошибке шины во время верификации прерывания от внешнего устройства.

Прерывания уровней 1-7.

Эти вектора содержат адреса подпрограмм-реакций на прерывания в соответствии с их приоритетами: если уровень, указанный в регистре статуса, больше уровня возникшего прерывания, то это прерывание игнорируется.

TRAP.

Генерируются командами TRAP (от TRAP #0 до TRAP #15).

Пользовательские вектора.

Эти вектора содержат адреса обработчиков прерываний от некоторых внешних устройств.

Мы не будем углубляться в детали механизма исключений, так как это выходит за рамки нашей книги. Ука-

ПРОЦЕССОР MC68000

жем лишь еще одну особенность: возврат из обработчиков исключений должен осуществляться по команде RTE (ReTurn from Exception). Эта команда корректно восстанавливает сохраненные в стеке регистры и возобновляет работу основной программы.

2.3.3 Прерывания.

Обработка прерываний происходит по той же схеме, что и обработка исключений. Прерывание — это нарушение нормального выполнения программы по внешнему сигналу.

Прерывания делятся на уровни — от 0 до 7, при этом прерывание уровня 7 является наиболее приоритетным. Как уже было отмечено, если возникает прерывание с уровнем, меньшим чем значение битов I0-I2 регистра SR, то это прерывание просто игнорируется. Иначе выполняются действия, как при генерации исключения (сохранение регистров SR, SP и PC и переход к обработчику, адрес которого берется из соответствующего уровню вектора (см. предыдущий пункт)).

Прерывания оказываются полезными для синхронизации программ с внешними устройствами (hardware). Например, при нажатии клавиш клавиатуры генерируется прерывание, обработчик которого распознает нажатую клавишу и вызывает соответствующие процедуры. Таким образом, в основной программе не нужно предусматривать проверку состояния клавиатуры, а кроме того, гарантируется быстрая и правильная реакция на нажатие клавиш. Аналогичный метод используется и при работе с интерфейсом RS232.

Подробнее о прерываниях мы поговорим в следующих разделах, а сейчас лишь напомним, что как и в случае исключений возврат из обработчика прерываний должен осуществляться по команде RTE.

По сути, прерывания и исключения — это одно и то же. Исключения — это прерывания, которые сигнализируют о какой-либо ошибке (например, деление на ноль).

ПРОЦЕССОР MC68000**2.3.4 Коды условий.**

Любой язык программирования поддерживает условные операторы. Например, строчка из BASIC'a

IF D1=2 THEN D2=0

является условным оператором. Чтобы переписать то же самое на ассемблере, нужно предварительно выполнить операцию сравнения:

CMP #2,D1 ;CMP – CoMPare (сравнить)

Как же работает сравнение на ассемблере?

Команды перехода (ветвления) в языке ассемблера проверяют выполнение условий из некоторого фиксированного набора. Этот набор условий кодируется битами 0-4 регистра статуса. Рассмотрим для примера бит 2 регистра SR: этот бит является флагом нулевого результата (устанавливается, если результат предыдущей команды — 0). Команда CMP устанавливает флаг Z, если ее операнды равны между собой (в действительности CMP производит вычитание первого операнда из второго и выставляет флаги условий в соответствии с результатом).

Таким образом, если в нашем примере D1 содержит число 2, то после команды CMP флаг Z будет установлен, и можно выполнить соответствующее ветвление:

```
...
CMP #2,D1 ;сравнение D1 с числом 2
BNE UNEQUAL ;ветвление, если не равно
              (флаг Z сброшен)
MOVE #0,D2 ;иначе выполнить D2=0
```

UNEQUAL

BNE (Branch if Not Equal) — это команда ветвления по неравенству (нулю). Это означает, что если в результате выполнения предыдущей команды (CMP) флаг Z оказывается сброшен (=0), то происходит передача управления (ветвление) по адресу, указанному в операнде команды.

ПРОЦЕССОР MC68000

Существует и в некотором смысле обратная команда — BEQ (Branch if Equal — ветвление по равенству), которая выполняет ветвление в случае $Z=1$.

Приведем список кодов условий, по которому можно легко построить любую команду ветвления формата Bcc (cc — код конкретного условия):

cc	Условие	Биты
T	всегда верно, аналог BRA	
F	всегда неверно, ветвления не происходит	
HI	выше	$C' * Z'$
LS	ниже или равно	$C + Z$
CC,HS	флаг C = 0 (выше или равно)	C'
CS,LO	флаг C = 1 (ниже)	C
NE	не равно	Z'
EQ	равно	Z
VC	флаг переполнения (V) = 0	V'
VS	флаг переполнения (V) = 1	V
PL	флаг знака (N) = 0 (плюс)	N'
MI	флаг знака (N) = 1 (минус)	N
GE	больше или равно	$N * V + N' * V'$
LT	меньше	$N * V' + N' * V$
GT	больше	$N * V * Z' + N' * V' * Z$
LE	меньше или равно	$Z + N * V' + N' * V$

* = логическое И,

+ = логическое ИЛИ,

' = логическое НЕ.

Приведем несколько примеров на применение условных ветвлений:

```
CMP #2,D1
BLS SMALLER_EQUAL
```

Здесь по команде BLS выполняется ветвление, если содержимое D1 ниже или равно двум, то есть $D1 = 0, 1$ или 2: команды BHI, BLS, BHS, BLO служат для сравне-

ПРОЦЕССОР MC68000

ния беззнаковых чисел (будем говорить, что число N1 ниже числа N2, если N1 меньше N2 в беззнаковом представлении, то есть без учета специального смысла старшего бита). Если же используется команда BLE, то ветвление будет корректно работать и для отрицательных чисел.

Команды BEQ и BNE могут применяться не только при сравнениях. Используя их можно, например, определять состояния некоторого набора битов заданного слова:

```
...
AND.B #%00001111,D1    ;проверяет биты по маске
BEQ  SMALLER            ;ветвление, если ни один
                        ;из четырех младших битов
                        ;не установлен

CMP.B #%00001111,D1
BEQ  ALL                 ;ветвление, если младшие
                        ;четыре бита установлены
```

Команда AND выполняет побитовое сравнение операндов по следующей схеме: если в какой-либо позиции биты обоих операндов установлены, то в соответствующий бит результата записывается 1, в противном случае — 0. Результат записывается на место второго операнда (в нашем примере D1), старое содержимое которого теряется. Таким образом, результат команды AND в нашем примере будет содержать в младших четырех битах копию соответствующих битов регистра D1, а в старших битах — нули. Такая техника называется маскированием (наложением маски). В приведенном нами примере команда AND выполняет маскирование младших четырех битов регистра D1. Конечно, Вы можете использовать любые другие битовые комбинации.

Если все биты результата команды AND нулевые, то устанавливается флаг нуля (Z-flag) и срабатывает ветвление BEQ. Иначе, по команде CMP производится сравнение этого результата с числом %00001111, и если имеет место

ПРОЦЕССОР MC68000

равенство (а это означает, что младшие 4 бита исходного содержимого D1 были установлены), то выполняется второе ветвление BEQ.

Другие коды условий тоже могут использоваться для разных целей. Забегая вперед скажем, что помимо использования с CMP, условия CC и CS могут служить для определения содержимого бита, “вытолкнутого” командами сдвига (ROL, ROR итд.).

Прежде чем перейти к описанию набора команд процессора MC68000, дадим еще одну полезную рекомендацию: С помощью пакета ассемблер + отладчик (например, Devras или AssemPro) можно легко исследовать влияние команд на флаги. Например, чтобы лучше понять, как работает команда CMP, введите в редактор системы следующую простую программу:

```
run: cmp    $10,d1  
      bra    run
```

Затем откомпилируйте этот текст и войдите в отладчик. Теперь, задавая различные значения D1 и используя команды пошагового выполнения, Вы сможете наблюдать результаты работы CMP по состоянию регистра SR.

Если Вы не до конца понимаете, как работает та или иная команда, то воспользуйтесь отладчиком: ведь сам процессор может “рассказать” о себе больше, чем любая книга.

2.4 Команды процессора MC680x0.

Итак, настало время перейти к описанию набора команд процессора MC680x0 (под MC680x0 мы будем понимать процессоры серии Motorola 68000, 68010, 68020 и старше). Сразу отметим, что ограничения на объем книги не позволят нам подробно описывать каждую команду, для этого существуют специальные книги (например, Programming the 68000 by Steve Williams).

Приводимые ниже таблицы содержат описания аргументов для каждой команды. Большинство систем (например, AssemPro) имеют встроенные help-таблицы по мето-

ПРОЦЕССОР MC68000

дам адресации и запросам системы, которыми Вы можете воспользоваться для получения дополнительной информации. Для описания операндов мы будем пользоваться следующими обозначениями:

Label - метка или адрес,
Reg(s) - регистр (регистры),
An - регистр адреса n,
Dn - регистр данных n,
Source - операнд источника,
Dest - операнд приемника,
<ea> - адрес или регистр (универсальный операнд),
#n - непосредственный операнд,
d8 - восьмибитное смещение,
d16 - шестнадцатитбитное смещение.

В таблицы включены также некоторые команды, работающие только на процессорах 68020 и выше (такие команды помечены как '68020+').

Управляющие команды.

Для начала рассмотрим список основных команд MC680x0, отвечающих за управление выполнением программы:

Мнемоника	Действие
Bcc Label	условное ветвление, зависит от кода условия cc
Bcc.S Label	условное короткое ветвление
BRA Label	безусловное ветвление
BRA.S Label	безусловное короткое ветвление
BSR Label	обращение к подпрограмме, адрес возврата запоминается в стеке
BSR.S Label	короткое обращение к подпрограмме
CHK <ea>,Dn	проверка содержимого регистра Dn на принадлежность диапазону 0..<ea>
DBcc Dn,Label	проверка условия, декремент и ветвление
DBRA Dn,Label	аналог DBF

ПРОЦЕССОР MC68000

JMP <ea>	безусловный переход (в отличие от BRA операнд задает абсолютный адрес перехода)
JSR <ea>	переход к подпрограмме (в отличие от BSR операнд задает абсолютный адрес подпрограммы)
NOP	нет операции
RESET	сброс внешних устройств (привилегированная команда)
RTD #offset	возврат из подпрограммы с восстановлением стека (68020+)
RTE	возврат из обработчика исключения, влияет на весь SR (привилегированная команда)
RTR	возврат с загрузкой флагов (влияет только на младший байт SR)
RTS	возврат из подпрограммы
Scc <ea>	запись в <ea> байта -1, если условие выполнено
STOP	останов процессора (привилегированная команда)
TRAP #n	генерация исключения по команде TRAP
TRAPV	проверка флага переполнения и генерация исключения

Сделаем несколько важных замечаний:

Когда происходит вызов подпрограммы (по BSR или JSR), адрес следующей за вызовом команды заносится в стек. По команде RTS этот адрес извлекается из стека и используется для возврата в основную программу.

Рассмотрим, к примеру, следующую программу:

```
run:  pea  subroutine ;адрес подпрограммы – в стек
      jsr  subroutine ;вызов подпрограммы
      move.l(sp)+,d1  ;выборка длинного слова из
                       ;стека
```

ПРОЦЕССОР MC68000

```
; illegal ;если нет отладчика — останов
subroutine:
    move.l(sp),d0 ;считываем адрес в d0
    rts ;и возврат
```

Первая команда, PEA, записывает адрес подпрограммы в стек. Затем, по команде JSR происходит вызов подпрограммы subroutine, при этом адрес возврата (адрес, с которого должна быть продолжена основная программа после выхода из подпрограммы) автоматически заносится в стек.

Внутри подпрограммы длинное слово, которое было последним занесено в стек, копируется в d0, после чего подпрограмма заканчивается командой rts.

Затем в основной программе происходит "выталкивание" длинного слова из стека в d1. Завершается программа директивой illegal (на случай, если используемая Вами система не имеет встроенного отладчика).

Откомпилируйте этот пример и войдите в отладчик. Теперь, пошагово выполняя программу и наблюдая за содержимым регистров D0 и D1, Вы увидите, что в D0 запишется адрес команды move.l(sp)+,d1 (адрес возврата из подпрограммы), а в D1 — адрес самой подпрограммы.

Команды ветвления (BRA, BSR, BEQ, итд.) отличаются от команд перехода (JMP, JSR) способом задания операнда: команды ветвления содержат расстояние от текущего значения программного счетчика (PC) до точки перехода, в то время как команды перехода содержат абсолютный адрес (32-битный операнд) для передачи управления. Любая команда ветвления (Vxx) имеет две формы: обычную (Vxx, Vxx.W) и короткую (Vxx.S) (MC68020 имеет еще и длинную (.L) форму ветвления). В обычной форме для относительного операнда отводится 16 бит, то есть ветвление может осуществляться на расстояния от -32768 до +32767 байт. В короткой форме операнд 8-битный, что позволяет задавать расстояния из диапазона -128..+127 байт (интерес-

ПРОЦЕССОР MC68000

но, почему расстояния измеряются в байтах а не в словах? Ведь при переходе на нечетный адрес ничего хорошего не будет, а допустимые расстояния ветвления можно было бы увеличить вдвое...). Команды JMP и JSR используются при длинных и абсолютных переходах (например при вызове подпрограмм операционной системы). Обратите внимание, что MC68000 не имеет команд условного абсолютного перехода (JNE, JEQ, итд.).

Команды DBcc (DBF, DBNE, DBEQ, итд.) используются для организации циклов. Рассмотрим, к примеру, команду DBF, которая работает так: сначала проверяется регистр (первый операнд), и если там ноль, то ветвление игнорируется. В противном случае из этого регистра вычитается единица и происходит ветвление по второму операнду. Типичный вид цикла DBF:

move	#counter-1,D0	:число итераций минус
		:один
loop:	:тело цикла
	
dbf	D0,loop	:если D0 не равен нулю,
		:декремент и ветвление

Обратите внимание, что начальное содержимое счетчика должно быть на единицу меньше, чем число повторений цикла (так как проверка счетчика производится ДО его уменьшения).

Другие команды DBcc используются для организации циклов с дополнительным условием. Например, DBEQ можно трактовать как "цикл, пока не ноль".

Примеры на применение команд DBcc мы рассмотрим в следующих разделах.

Команды STOP и RESET являются привелегированными, но даже в режиме супервизора не рекомендуется их использовать (это может привести к "зависанию").

Команды TRAP имеют параметр — номер генерируемого TRAP-исключения, который определяет адрес век-

ПРОЦЕССОР MC68000

тора исключения (из диапазона \$0080-\$00BF). Некоторые операционные системы для 680x0 используют TRAP-векторы для своих запросов, но об этом мы поговорим позже.

Арифметические команды.

Вспомним пример с командой CMP, которую мы использовали для сравнения содержимого регистра данных с некоторым числом. Напомним, что команда CMP производит вычитание первого операнда из второго и на основании полученного результата выставляет флаги условий, причем ни один из операндов не изменяется. MC680x0 также имеет и специальную команду для вычитания — SUB, отличие которой от CMP состоит в том, что в случае SUB результат вычитания записывается во второй операнд, старое содержимое которого теряется. Существует и аналогичная команда для сложения — ADD. Во многих восьмиразрядных процессорах (например, в 6502) этим и ограничивается набор арифметических операций, в то время как MC680x0 может еще умножать, делить и производить ряд других действий над целочисленными операндами.

Большинство арифметических команд процессора MC680x0 требуют два аргумента. Например, в команде

SUB source,dest

операнд source задает вычитаемое, а операнд dest — уменьшаемое. При этом результат операции всегда записывается по адресу второго операнда.

Приведем таблицу основных арифметических операций над целыми числами:

Мнемоника		Действие
ADD	<ea>,Dn	сложение операнда <ea> с регистром данных
ADD	Dn,<ea>	сложение регистра Dn с операндом <ea>; результат записывается по адресу <ea>
ADDA	<ea>,An	сложение с адресным регистром
ADDI	#n,<ea>	сложение с непосредственным

ПРОЦЕССОР MC68000

	операндом
ADDQ #n,<ea>	быстрое сложение с константой из интервала 1-8
ADDX Dn,Dn	сложение регистров данных и переноса (флаг X)
ADDX -(An),-(An)	сложение с переносом в памяти, работает только с предкрементной адресацией
CLR <ea>	обнуление операнда
CMP <ea>,Dn	сравнение с регистром данных
CMPA <ea>,An	сравнение с регистром адреса
CMPI #n,<ea>	сравнение с непосредственным операндом
CMPM (An)+,(An)+	сравнение в памяти, работает только с постинкрементной адресацией
CMP2 <ea>,Rn	сравнить регистр с верхней и нижней границами (68020+)
DIVS <ea>,Dn	знаковое деление регистра Dn на 16-битный операнд; частное записывается в младшее слово регистра, а остаток — в старшее. Если частное не умещается в одно слово, результат теряется
DIVS.L <ea>,Dn	то же, но оба операнда — 32-битные (68020+)
DIVS.L <ea>,Dn:Dn	то же, но результат и остаток — 32-битные, а делимое — 64-битное (68020+)
DIVSL.L <ea>,Dn:Dn	то же, но все операнды и результат — 32-битные (68020+)
DIVU <ea>,Dn	беззнаковое деление, работает аналогично DIVS
DIVU.L <ea>,Dn	см. DIVS.L <ea>,Dn (68020+)
DIVU.L <ea>,Dn:Dn	см. DIVS.L <ea>,Dn:Dn (68020+)
DIVUL.L <ea>,Dn:Dn	см. DIVSL.L <ea>,Dn:Dn (68020+)
EXT Dn	расширение знака (в случае .W — расширение байта до слова, в случае

ПРОЦЕССОР MC68000

		.L — слова до длинного слова)
EXTB.L Dn		расширение байта до длинного слова (68020+)
MULS <ea>,Dn		знаковое умножение регистра Dn на операнд <ea> (оба множителя рассматриваются как 16-битные целые), результат имеет размер длинного слова
MULS.L <ea>,Dn		то же, только делитель рассматривается как длинное слово (68020+)
MULS.L <ea>,Dn:Dn		то же, только результат имеет размер 64 бита (68020+)
MULU <ea>,Dn		беззнаковое умножение, функционирует аналогично MULS
MULU.L <ea>,Dn		см. MULS.L <ea>,Dn (68020+)
MULU.L <ea>,Dn:Dn		см. MULS.L <ea>,Dn:Dn (68020+)
NEG <ea>		изменение знака числа (вычитание из нуля)
NEGX <ea>		изменение знака с вычитанием переноса
SUB <ea>,Dn		вычитание операнда <ea> из регистра данных
SUB Dn,<ea>		вычитание регистра данных из <ea>
SUBA <ea>,An		вычитание из адресного регистра
SUBI #n,<ea>		вычитание непосредственного операнда
SUBQ #n,<ea>		быстрое вычитание 3-битовой константы
SUBX Dn,Dn		вычитание регистра и переноса
SUBX -(An),-(An)		вычитание в памяти с переносом (работает только с предкрементной адресацией)
TST <ea>		сравнить операнд с нулем и выставить флаги

ПРОЦЕССОР MC68000

Обратите внимание на то, что для выполнения одной и той же операции может существовать несколько команд в зависимости от способа адресации операндов (например, ADD, ADDI, ADDA итд.). Однако ассемблер допускает записи типа ADD D0, A0, автоматически подставляя подходящую команду (в данном случае ADDA). Сказанное не относится к командам ADDQ и SUBQ: их надо указывать явно (даже если непосредственный операнд уместается в 3 бита, ассемблер не будет заменять ADD на ADDQ). Команды ADDQ и SUBQ выполняются с такой же скоростью, что и операции над регистрами (четыре такта на MC68000 и до двух тактов на MC68020), причем непосредственный операнд размещается в первом (и единственном) слове кода команды. Таким образом, команды ADDQ и SUBQ занимают меньше памяти (на два байта), чем ADDI или SUBI.

Как видно из таблицы, арифметические команды допускают не все способы адресации операндов. Например, нельзя одной командой выполнить сложение операндов в памяти (то есть запись ADD VAL1, VAL2 является недопустимой). Существуют и другие ограничения:

если <ea> — операнд приемника, то недопустима косвенная адресация через программный счетчик (PC) (это относится ко всем командам MC680x0);

операции с адресными регистрами не могут иметь размер .B, а в случае .W операнды расширяются до .L с учетом знака (дополняются до 32 бит нулями или единицами, в зависимости от знака операндов)..

Процессор MC680x0 также имеет ряд команд для работы с двоично-десятичными (BCD) числами. BCD-число — это набор полубайтов, в каждом из которых содержатся цифры от 0 до 9, а остальные значения (от A до F) просто не используются. Например, десятичное число 2891 представляется как BCD-число:

$$\begin{array}{ccccccc} \%0010 & + & \%1000 & + & \%1001 & + & \%0001 & = & \%0010100010010001 \\ 2 & & 8 & & 9 & & 1 & & \end{array}$$

Закодированные таким образом десятичные числа

ПРОЦЕССОР MC68000

легко складывать и вычитать на аппаратном уровне. MC680x0 поддерживает следующие операции над BCD-числами:

Мнемоника	Действие
ABCD Dn,Dn	сложение BCD-чисел с переносом (флаг X)
ABCD -(An),-(An)	то же NBCD <ea> изменение знака BCD-числа
SBCD Dn,Dn	вычитание BCD-чисел с переносом (флаг X)
SBCD -(An),-(An)	то же PACK Dn,Dn,#n преобразование чисел из формата 'байт на цифру' в формат BCD (68020+)
PACK -(An),-(An),#n	то же (68020+)
UNPK Dn,Dn,#n	преобразование BCD-чисел в формат 'байт на цифру' (68020+)
UNPK -(An),-(An),#n	то же (68020+)

Обратите внимание, что для команд ABCD и SBCD допустимы только два метода адресации: регистровая и предекрементная.

Далее мы рассмотрим логические операции и операции для работы с отдельными битами:

Мнемоника	Действие
AND <ea>,Dn	побитовое логическое И
AND Dn,<ea>	то же
ANDI #n,<ea>	побитовое логическое И с константой
EOR Dn,<ea>	побитовое исключающее ИЛИ
EORI #n,<ea>	побитовое исключающее ИЛИ с константой
NOT <ea>	инвертирование (побитовое НЕ)
OR <ea>,Dn	побитовое логическое ИЛИ
OR Dn,<ea>	то же
ORI #n,<ea>	побитовое логическое ИЛИ с

ПРОЦЕССОР MC68000

TAS <ea> константой
 проверка байта и установка 7-го бита

Отдельными битами можно манипулировать используя следующие команды:

Мнемоника	Действие
BCHG #n,<ea>	инвертирование бита n (0 заменяется на 1, 1 — на 0)
BCHG Dn,<ea>	то же (номер бита задается в регистре Dn)
BCLR #n,<ea>	очистка (обнуление) бита n
BCLR Dn,<ea>	то же
BSET #n,<ea>	установка бита n (запись 1)
BSET Dn,<ea>	то же
BTST #n,<ea>	сравнить бит n с нулем и выставить Z-флаг
BTST Dn,<ea>	то же

Следует помнить, что номер бита задается по модулю 32, если <ea> — это регистр данных, и по модулю 8, если <ea> задает адрес в памяти. При этом допустимыми размерами битовых операций (Bxxx) являются .B и .L.

И еще одно важное замечание: в логических командах в качестве обобщенного операнда <ea> не может фигурировать адресный регистр.

Процессор MC68020 имеет команды для работы не только с отдельными битами, но и с битовыми полями. Эти команды подробно описаны в справочном руководстве процессора MC68020.

Эти команды особенно полезны при работе с регистрами внешних устройств, об этом мы поговорим подробнее в следующих разделах.

Следующий класс команд — команды сдвига, которые позволяют "сдвигать" операнд (в пределах своего битового поля) на произвольное число битов. Сначала приведем список этих команд, а затем рассмотрим их работу бо-

ПРОЦЕССОР MC68000

более подробно:

Мнемоника	Действие
ASL Source, Dest	арифметический сдвиг влево на n бит
ASR Source, Dest	арифметический сдвиг вправо на n бит
LSL Source, Dest	логический сдвиг влево на n бит
LSR Source, Dest	логический сдвиг вправо на n бит
ROL Source, Dest	прокрутка влево на n бит
ROR Source, Dest	прокрутка вправо на n бит
ROXL Source, Dest	прокрутка на n бит влево через перенос
ROXR Source, Dest	прокрутка на n бит вправо через перенос

Допустимые значения операндов Source и Dest:

Source	Dest
Dn	Dn
#n	Dn
—	<ea> (кроме регистров)

Операнд Source определяет, на сколько битов сдвигать операнд Dest (когда в качестве Dest выступает адрес в памяти, возможен сдвиг только на один бит).

На самом деле операции арифметического сдвига на n бит влево/вправо эквивалентны умножению/делению операнда на два в степени n. Поясним это на небольшом примере.

Рассмотрим байт, содержащий число 16 (или %00010000 в двоичном представлении). Что произойдет если над этим байтом выполнить операцию сдвига влево? Имеем:

%00010000 = 16 (исходное число)

%00100000 = 32 (сдвинутое на один бит влево)

Каждый последующий сдвиг удваивает операнд, поэтому после n сдвигов имеем исходное значение, умноженное на 2^n (пока рассматриваем ситуацию, когда биты не выдвигаются за разрядную сетку).

Аналогично, при сдвиге вправо происходит деление

ПРОЦЕССОР MC68000

на 2^n , но здесь есть одна особенность. Например, сдвинем вправо число 5:

$\%00000101 = 5$ (исходное число)

$\%00000010 = 2$ (сдвинутое на один бит вправо)

В результате сдвига получаем 2, а не 2.5 — ведь мы имеем дело только с целыми числами. Как видно из примера, при арифметическом сдвиге могут теряться отдельные биты (выдвигаемые за разрядную сетку).

Итак, с помощью команд сдвига можно умножать числа на степени двойки. При этом сдвиги выполняются значительно быстрее, чем команды деления и умножения (DIVx и MULx).

Теперь поясним, зачем нужно столько модификаций команд сдвига (ASR, LSR итд.). Прежде всего рассмотрим отличие сдвига от прокрутки.

При логическом сдвиге операнда к нему справа или слева добавляется нулевой бит, в то время как при прокрутке добавится бит, “вытолкнутый” за разрядную сетку. Таким образом, прокрутка (ее часто называют циклическим сдвигом) обеспечивает сохранение всех битов слова (байта): то, что “выталкивается” с одной стороны, добавляется с другой. Такая прокрутка реализуется командами ROR и ROL. Команды ROXR и ROXL работают несколько по-другому: то, что “выталкивается” за разрядную сетку, попадает в X-флаг (см. описание регистра SR), а старое его содержимое добавляется с другой стороны операнда (происходит прокрутка через X-флаг, который играет роль дополнительного бита сдвигаемого слова (байта)).

Более наглядно схему работы сдвигов и прокруток можно представить так:

LSR: 0 → ОПЕРАНД → C, X

ROR: → ОПЕРАНД → C, X

ROXR: → ОПЕРАНД → C, X →

ПРОЦЕССОР MC68000

Сдвиги в свою очередь делятся на арифметические и логические. В случае логического сдвига, к слову (байту) всегда добавляется нулевой бит, а выдвинутый с другой стороны бит записывается в С и Х-флаги. Арифметические сдвиги позволяют выполнять умножение/деление операнда на 2^n с учетом знака. Так, команда ASR не трогает старший бит сдвигаемого операнда, сохраняя таким образом знак результата, а команда ASL полностью аналогична команде LSL:

ASR:  ОПЕРАНД → C,X

ASL,LSL: C,X ← ОПЕРАНД ← 0

И наконец, рассмотрим команды пересылки данных, которые, по всей видимости, составляют основу набора команд любого процессора.

Мнемоника		Действие
EXG	Rn,Rn	обмен значениями двух регистров
LEA	<ea>,An	загрузка адреса операнда <ea> в регистр An; в качестве <ea> не могут быть указаны регистры; не допускаются постинкрементный и предекрементный методы адресации
LINK	An,#n	связать стековый фрейм с An
MOVE	<ea>,<ea>	пересылка (копирование)
MOVE	SR,<ea>	чтение регистра состояния (SR) (привелегированная команда)
MOVE	<ea>,SR	запись в регистр состояния (SR) (привелегированная команда)
MOVE	USP,<ea>	чтение указателя стека пользователя (привелегированная команда)
MOVEA	<ea>,An	пересылка в адресный регистр
MOVEM	Regs,<ea>	запись набора регистров в память
MOVEM	<ea>,Regs	чтение набора регистров из памяти
MOVEP	Dn,d16(An)	запись во внешний регистр

ПРОЦЕССОР MC68000

MOVEP d16(An),Dn	чтение из внешнего регистра
MOVEQ #n,Dn	быстрая пересылка восьмибитной константы в Dn с расширением знака до .L
PEA <ea>	размещение адреса операнда <ea> в стеке
SWAP Dn	обмен значениями старшего и младшего слов регистра Dn
UNLK An	отсоединить стековый фрейм от An

Сделаем несколько замечаний:

Команда LEA может использоваться для загрузки адреса переменной в регистр, например, команда LEA Label,A0 эквивалентна команде MOVE #Label,A0. Обратите внимание на различие синтаксиса этих команд: в случае LEA первый операнд — это сама переменная, а в случае MOVE — непосредственное значение адреса метки Label. Команда LEA не допускает использования непосредственного, регистрового, постинкрементного и предкрементного методов адресации.

Но этим не ограничивается применение команды LEA, рассмотрим более интересный случай:

LEA 1(A0,D0),A1

Что будет записано в регистр A1 после выполнения этой команды? Ответ очевиден: адрес первого операнда, который вычисляется как

$1 + \text{<содержимое A0>} + \text{<содержимое D0>}$

(см. п 2.2). Того же результата можно добиться и с помощью команды MOVE:

```
MOVE.L    A0,A1
ADD.L     D0,A1
ADDQ.L    #1,A1,
```

только в этом случае мы имеем на две команды больше. Как Вы можете видеть, команда LEA дает весьма интересные возможности.

Аналогичные возможности дает и команда PEA, ко-

ПРОЦЕССОР MC68000

торая записывает адрес своего операнда в стек. При этом действуют те же ограничения на методы адресации, что и в случае LEA.

Команды LINK и UNLK служат для упрощения организации локального стека в пользовательских функциях. Часто локальные (временные) параметры хранят в стеке, используя некоторый адресный регистр как указатель на начало области параметров (эта область называется фреймом (frame)). При помощи команды LINK можно выделить фрейм для локальных переменных в стеке. Пусть, например, начальное значение SP равно \$1000, а A0=\$12345678. Тогда по команде LINK A0,#-4 произойдет следующее: содержимое A0 запишется в стек, в A0 запишется содержимое SP, а сам SP уменьшится на 4. В результате стек будет выглядеть так:

```
SP → $0FFC (B)   : ???  
          $0FFE (A) : ???  
A0 → $1000       : $12345678
```

SP, как обычно, указывает на последнее слово стека, а в A0 хранится прежнее содержимое SP. Доступ к переменным в этом случае может осуществляться через регистр A0. Например, для копирования значения A в B используется команда MOVE -2(A0),-4(A0).

Для освобождения фрейма используется команда UNLK, которая работает так: сначала в SP копируется содержимое A0, а затем из стека восстанавливается прежнее содержимое A0.

Команда MOVEM используется в основном для сохранения (восстановления) некоторого набора регистров в стеке (из стека). Например, команда

```
MOVEM.L D0-D7/A0-A6,{SP}
```

записывает содержимое всех регистров (кроме A7) в стек, а команда

```
MOVE.L (SP)+,D0-D7/A0-A6
```

восстанавливает значения регистров из стека. В общем случае операнд Regs задает список регистров для пересылки:

ПРОЦЕССОР MC68000

отдельные регистры записываются через разделитель '/', а диапазоны регистров — через '-' (например, запись D0/D2-D4/A0-A3 эквивалентна D0/D2/D3/D4/A0/A1/A2/A3).

Команда MOVEM может иметь размер .W или .L. При использовании MOVE.W <ea>, Regs каждый регистр из списка Regs будет дополняться до длинного слова с расширением знака.

Команда MOVEM <ea>, Regs не допускает использование предкрементного метода адресации, а команда

MOVEM Regs, <ea> — постинкрементного, например:

MOVEM -(A0), D0/D1 ; недопустимо

MOVEM D0/D1, (A0)+ ; недопустимо

Итак, мы рассмотрели почти все команды процессора MC680x0. Как видно из таблиц, процессор предоставляет достаточно мощные средства для обработки данных. Используя различные методы адресации можно писать очень эффективные ассемблерные программы: именно гибкость и эффективность выгодно отличают ассемблер MC680x0 от ассемблеров других архитектур (например, Intel).

Завершим описание команд MC680x0 обобщенной таблицей, в которой приведены допустимые методы адресации команд и их влияние на флаги условий (дополнительные команды MC68020 в таблицу не включены). Будем использовать следующие обозначения:

x - допустимо

s - допустимо, только если <ea> — операнд источника

d - допустимо, только если <ea> — операнд приемника

- - не изменяется

0 - обнуляется

Мнемоника	1	2	3	4	5	6	7	8	9	10	11	12		XNZVC P
ABCD	x				x									
ADD	x	s	x	x	x	x	x	x	x	s	s	s		*****
ADDA	x	x	x	x	x	x	x	x	x	x	x	x		-----
ADDI	x		x	x	x	x	x	x	x					*****
ADDQ	x	x	x	x	x	x	x	x	x					*****
ADDX	x					x								*****

AMIGA: ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ.

ПРОЦЕССОР MC68000												
	1	2	3	4	5	6	7	8	9	10	11	12
AND	x		x	x	x	x	x	x	x	s	s	s
ANDI	x		x	x	x	x	x	x	x			
ASL,ASR	x		x	x	x	x	x	x	x			
Bcc												
BCHG	x		x	x	x	x	x	x	x			
BCLR	x		x	x	x	x	x	x	x			
BRA												
BSET	x		x	x	x	x	x	x	x			
BSR												
BTST	x		x	x	x	x	x	x	x	x	x	x
CHK	x		x	x	x	x	x	x	x	x	x	x
CLR	x		x	x	x	x	x	x	x			
CMF	x	x	x	x	x	x	x	x	x	x	x	x
CMFA	x	x	x	x	x	x	x	x	x	x	x	x
CMPI	x		x	x	x	x	x	x	x			
CMPM				x								
DBcc												
DIUS	x		x	x	x	x	x	x	x	x	x	x
DIUJ	x		x	x	x	x	x	x	x	x	x	x
EOR	x		x	x	x	x	x	x	x			
EORI	x		x	x	x	x	x	x	x			
EORI CCR												
EORI SR												
EXG	x	x										
EXT	x											
EXTB	x											
ILLEGAL												
JMP				x			x	x	x	x	x	x
JSR				x			x	x	x	x	x	x
LEA				x			x	x	x	x	x	x
LINK		x										
LSL,LSR	x		x	x	x	x	x	x	x			
MOVE	x	s	x	x	x	x	x	x	x	s	s	s
MOVEA	x	x	x	x	x	x	x	x	x	x	x	x
MOVE from SR	x		x	x	x	x	x	x	x			
MOVE to SR	x		x	x	x	x	x	x	x	x	x	x
MOVE USP		x										
MOVEM				x	s	d	x	x	x	x	s	s
MOVEP	s	d										
MOVEQ	d											
MULS	x		x	x	x	x	x	x	x	x	x	x
MULU	x		x	x	x	x	x	x	x	x	x	x
NBCD	x		x	x	x	x	x	x	x			
NEG	x		x	x	x	x	x	x	x			
--												
	1	2	3	4	5	6	7	8	9	10	11	12

xNzVC P

---00

---00

---0000

---00

---00

AMIGA: ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ.

ПРОЦЕССОР MC68000													XNZVC	P
	1	2	3	4	5	6	7	8	9	10	11	12		
NEGX	x		x	x	x	x	x	x	x				-----	
NOP													-----	
NOT	x		x	x	x	x	x	x	x				---00	
OR	x		x	x	x	x	x	x	x	s	s	s	---00	
ORI	x		x	x	x	x	x	x	x				---00	
PEA			x			x	x	x	x			x	-----	
RESET													-----	P
ROL, ROR	x		x	x	x	x	x	x	x				---00	
ROXL, ROXR	x		x	x	x	x	x	x	x				---00	
RTE													-----	P
RTR													-----	
RTS													-----	
SBCD	x				x								---u---	
Sec	x		x	x	x	x	x	x	x				-----	
STOP													-----	P
SUB	s	s	x	x	x	x	x	x	x	s	s	s	-----	
SUBA	x	x	x	x	x	x	x	x	x	x	x	x	-----	
SUBI	x		x	x	x	x	x	x	x				-----	
SUBQ	x	x	x	x	x	x	x	x	x				-----	
SUBX	x				x								-----	
SWAP	x												---00	
TAS	x		x	x	x	x	x	x	x				---00	
TRAP										x			-----	
TRAPV													-----	
IST	x		x	x	x	x	x	x	x				---00	
UNLK		x											-----	

* - изменяется в соответствии с состоянием условия

l - устанавливается

u - неопределено

P - привилегированная команда

Раздел 3. СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ.

Как мы уже говорили, для выполнения любой ассемблерной программы необходимо перевести ее в двоичное представление, понятное процессору. Для такого перевода используются специальные программы — ассемблеры (assemblers). На Amiga существует множество систем программирования, в состав которых входят ассемблеры. О том, как работать с такими системами, мы и поговорим в этом разделе, а именно, мы рассмотрим три наиболее популярные программы:

ASSEM

Это ассемблер, входящий в базовый пакет программ Amiga. ASSEM предоставляет достаточно широкие возможности, однако он не имеет встроенного отладчика.

AssemPro

Это ассемблер фирмы Abacus, имеющий встроенный отладчик. AssemPro является одной из наиболее популярных систем программирования на Amiga, и в связи с этим большая часть программ, приводимых в этой книге, написана именно для AssemPro.

KUMA-SEKA

Это еще один достаточно удобный ассемблер со встроенным отладчиком.

С помощью любого из этих ассемблеров Вы можете транслировать свои программы и записывать на диск исполняемые файлы.

3.1. Ассемблер ASSEM

ASSEM — это обычный дисковый ассемблер, который

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

способен транслировать ассемблерные тексты из файлов и записывать на диск объектные модули. Он не имеет встроенного текстового редактора для ввода программ.

ASSEM может быть запущен из CLI с указанием параметров в командной строке, например::

ASSEM Source -O Destination

Source — это имя файла, содержащего текст программы. Destination — имя файла для записи результатов трансляции. Опция “-O” говорит о том, что на выходе ассемблера должен быть объектный модуль.

Приведем полный список опций ассемблера ASSEM (опции указываются перед именем выходного файла):

- O получить объектный модуль;
- V записывать все сообщения об ошибках трансляции в файл (если эта опция не указана, то сообщения об ошибках будут выводиться на экран);
- L выводить текст программы в файл (для распечатки на принтер можно использовать PRT:);
- H добавить указанный файл к исходному тексту для трансляции;
- E создать файл, содержащий директивы EQU;
- C (или OPT) указание дополнительных опций:
 - OPT S - создать символьную таблицу меток и их адресов,
 - OPT Z - создать таблицу перекрестных ссылок,
 - OPT W - выделить рабочую область.

Ассемблер создает некий промежуточный файл (объектный модуль), который не является исполняемым. Чтобы получить из него файл, готовый к выполнению, нужно использовать компоновщик ALINK. С помощью компоновщика можно объединять несколько объектных модулей для создания одного исполняемого файла. В простейшем случае вызов программы ALINK выглядит так:

ALINK Source TO Destination

Source — имя объектного модуля, созданного ассемблером,

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

Destination — имя выходного исполняемого файла.

3.2. Система AssemPro

AssemPro — это интегрированная среда программирования. Это означает, что в состав этой системы входит несколько компонент (редактор, ассемблер и отладчик), использующих единый интерфейс.

AssemPro использует несколько окон: по одному на каждую компоненту системы и окно подсказок. Для создания программы нужно:

- 1) Написать текст программы в редакторе и сохранить его на диске.
- 2) С помощью ассемблера перевести программу в машинный код.
- 3) Отладить программу, используя встроенный отладчик.

Отладчик позволяет тестировать фрагменты программы и даже отдельные команды. После каждого шага Вы можете визуальным образом наблюдать текущее состояние регистров и флагов условий, что значительно облегчает поиск ошибок в программе.

Для того, чтобы начать работу с AssemPro, Вам нужно запустить всего один файл: все компоненты системы загружаются вместе. Таким образом, Вам не потребуется сохранять на диске промежуточные файлы редактора, ассемблера и компоновщика.

AssemPro имеет еще одну полезную компоненту — реассемблер. С помощью реассемблера можно восстанавливать исходный текст программы по ее коду. Полученный текст можно отредактировать с помощью встроенного редактора и оттранслировать заново.

При работе с AssemPro, имейте в виду следующее: для выравнивания по четности в AssemPro используется директива ALIGN, а не EVEN, как в системе K-SEKA. Еще одной особенностью AssemPro является то, что исходные тексты программ обязательно должны завершаться директивой END.

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

Далее приведем небольшое руководство по работе с AssemPro.

Запустив AssemPro Вы можете либо начать ввод новой программы в окне текстового редактора, либо загрузить уже готовый текст с диска: для этого нужно выбрать соответствующий пункт меню или нажать клавиши <AMIGA> и <o>. Далее Вы сможете выбрать нужный файл в окне файл-реквестера (requester).

После того, как Вы ввели (или загрузили с диска) текст программы, можно выполнить трансляцию (ассемблирование), но перед этим рекомендуется записать исходный текст на диск. Трансляция запускается нажатием клавиш <AMIGA> и <a> в окне ассемблера, которое располагается над окном редактора.

Перед началом трансляции Вам будет предложено выбрать тип памяти для размещения кода. Помните, что данные для custom-чипов должны располагаться в CHIP-памяти.

Итак, по нажатию ОК начинается ассемблирование Вашей программы. Если дополнительно выбрать опцию "breakable", то процесс трансляции можно будет прервать в любой момент нажатием обеих клавиш Shift. Все ошибки будут выводиться в специальное окно, используя которое Вы сможете исправить текст и продолжить трансляцию, выбрав "Save and try again" (записать и повторить).

После трансляции в памяти Amiga будет находиться готовая к выполнению машинная программа. Используя пункт меню "save as" Вы сможете записать исполняемый модуль на диск (или на RAM диск, написав RAM: перед именем файла). В дополнении к исполняемому файлу можно записать и пиктограмму (icon) для последующего запуска Вашей программы из системы Workbench.

Для тестирования полученной программы активизируйте окно отладчика и выберите пункт меню "Load" (или нажмите <AMIGA> <o>) для загрузки отлаживаемой программы. После выбора нужного файла в файл-реквестере

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

Ваша программа загрузится в память и появится в окне отладчика в виде ассемблерного текста.

Подсвеченная строка указывает текущее положение программного счетчика (PC). При запуске программы первой выполнится команда, записанная именно в этой строке. Существует три способа заставить программу выполняться: Первый способ состоит в выборе пункта "Start", но в этом случае Вы не сможете прервать программу до ее завершения.

Второй вариант — "Start breakable" — в этом отношении лучше, он позволяет прервать выполнение программы в любой момент по нажатию клавиши <Esc> (это работает только в том случае, если Ваша программа сама не использует <Esc>). Кроме того, во время работы программы отладчик постоянно выводит текущее содержимое регистров в левой части окна.

Третий вариант запуска программы в отладчике — запуск по частям. Это делается либо с помощью пошагового выполнения, либо расстановкой контрольных точек (breakpoints). Контрольные точки ставятся нажатием <AMIGA> после выбора нужного места с помощью мыши. Если Вы теперь запустите программу, то она прервется, как только управление дойдет до контрольной точки.

Отладчик AssemPro имеет еще одну полезную возможность для тестирования фрагментов программ: Вы можете с помощью мыши выделить часть программы (подобно тому, как в текстовых редакторах выделяются блоки текста) от текущей команды до нужного места остановки, после чего отладчик выполнит выделенный фрагмент. Эта функция отладчика оказывается очень полезной при пошаговом тестировании программ.

AssemPro имеет еще одно полезное окно — окно подсказок. В этом окне выводятся таблицы возможных методов адресации команд (нужно просто указать мышью на интересующую Вас команду), а также параметры вызовов

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

операционной системы. Так что работая с AssemPro, Вам не придется листать документацию по командам процессора в поисках нужной таблицы.

3.3. Система K-SEKA.

Система SEKA (от KUMA), помимо ассемблера, содержит простой текстовый редактор и отладчик. Эта система управляется запросами, вводимыми из командной строки, и является достаточно удобным и гибким средством для создания программ. Правда, Вам потребуется сначала привыкнуть к редактору SEKA, который может на первый взгляд показаться сложным в обращении.

Чтобы загрузить текст программы из файла в редактор, нажмите "r" (read). На запрос системы "FILENAME>" нужно ввести имя файла для загрузки. Имена текстовых файлов созданных с помощью SEKA, имеют расширение ".S" (от Source — исходный), но при загрузке таких файлов Вам достаточно ввести имя без указания ".S": система сама подставит расширение.

Записать модифицированный текст программы можно с помощью команды "w" (write). Если на запрос имени файла ввести, например, "Test", то система запишет на диск файл с именем "Test.S". Это обычный текстовый файл в ASCII формате.

Изменять текст программы можно используя один из двух редакторов: строковый или экранный. Экранный редактор вызывается нажатием <Esc>, после чего верхняя часть экрана выделяется для редактирования текста. Вы можете свободно перемещать курсор по тексту и добавлять новые строки, которые автоматически пронумеровываются. Оканчивается текст любой программы словом <END>. Выйти из редактора можно повторным нажатием клавиши <Esc>.

Другие запросы SEKA вводятся из командной строки с подсказкой "SEKA>" (командный режим). Далее мы приведем список запросов для редактирования текста:

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

Запрос	Действие
t	Поместить курсор на первую строку текста
t <n>	Поместить курсор на строку с номером n
b	Поместить курсор на последнюю строку текста
u	Перейти на одну строку вверх
u <n>	Перейти на n строк вверх
d	Перейти на одну строку вниз
d <n>	Перейти на n строк вниз
z	Удалить текущую строку
z <n>	Удалить n строк, начиная от текущей
e	Отредактировать текущую строку
e <n>	Редактировать, начиная со строки с номером n
ftext	Искать вхождение слова text, начиная с текущей строки. Заглавные буквы отличаются от строчных; пробелы после "f" относятся к слову для поиска, так что будьте внимательны
f	Повторить поиск слова
i	Запустить строковый редактор, с помощью которого можно вводить текст построчно. Вводимые строки автоматически нумеруются
ks	Уничтожить исходный текст. Перед уничтожением система выдаст запрос "sure?" ("Вы уверены?")
o	Отменить предыдущий запрос "ks" и восстановить старый текст
p	Напечатать текущую строку
p <n>	Напечатать n строк, начиная с текущей.

Совместное использование этих запросов и экранного редактора позволяет достаточно легко вводить и изменять исходные тексты. Например, если Вам требуется удалить строку в экранном редакторе, нужно подвести курсор к этой строке, выйти в командный режим по <Esc> и ввести запрос "z".

Приведем другой пример: пусть Вам нужно отредак-

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

тировать все строки, содержащие слово "trap". Для этого:

- ♦ переместите курсор на начало текста запросом "t";
- ♦ выполните поиск слова "trap" с помощью запроса "fttrap";
- ♦ нажмите <Esc> и отредактируйте строку;
- ♦ еще раз нажмите <Esc> для возврата в командный режим;
- ♦ используйте "f" для повторного поиска, итд.

По началу это выглядит весьма неуклюже, однако на практике работает весьма неплохо, и главное, быстро. Немного тренировки — и Вы привыкнете к редактору SEKA.

Теперь рассмотрим запросы для работы с диском:

Запрос	Действие
v	Вывести список файлов в текущем каталоге. Можно указать любое другое устройство или подкаталог: например, запрос "vc" распечатает содержимое каталога "c", причем этот каталог станет текущим
kf	Удалить файл (имя запрашивается после ввода запроса). <i>Будьте осторожны! Система не спросит подтверждения на удаление файла.</i>
r	Загрузить текстовый файл
ri	Загрузить двоичный файл в память. Имя файла, начальный адрес и верхняя граница памяти запрашиваются сразу после ввода "ri".
rx	Работает аналогично "ri", только файл загружается не с диска, а из последовательного порта (имя файла не запрашивается)
rl	Загрузить файл связей SEKA. Система выдаст запрос на подтверждение, так как после загрузки файла связей старое содержимое буфера текста теряется
w	Записать исходный текст на диск
wi	Записать блок памяти на диск; начальный и конечный адреса запрашиваются специально
wx	Передать блок памяти через последовательный порт
wl	Записать файл связей на диск. Запрос работает

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

только после трансляции программы с опцией l. Сообщение "*** Link option not specified" сигнализирует о невозможности создания файла связей

После загрузки или ввода текста программы Вы можете выполнить трансляцию: для этого используется директива "a". Перед началом трансляции система выведет запрос опций ассемблера. Если просто нажать <Enter>, программа будет оттранслирована обычным способом — результат сохранится в памяти.

Кроме этого можно использовать следующие опции:

- v вывести результат на экран
- p вывести результат на принтер
- h запрашивать подтверждение после вывода каждой страницы
- o использовать оптимизатор ветвлений, который добавляет суффиксы "S" там, где это возможно. В результате код программы получается компактнее
- l создать файл связей, который можно в последствии записать командой "wl"

При желании можно включить в листинг символьную таблицу, которая содержит имена и адреса меток, а также макросы. Макросы позволяют объединять несколько команд в одну псевдокоманду (макрокоманду).

Пусть, например, имеется подпрограмма, выводящая текст, на который указывает A0. Каждый раз при вызове этой подпрограммы нам нужно писать примерно следующее:

```
lea    text,A0      ;загрузка адреса текста в A0
bsr    pline         ;вывод текста
```

С помощью макросов можно упростить эту процедуру. Для этого в начале программы нужно написать:

```
print: macro ;начало описания макроса print
lea    ?1,A0 ;параметр — в A0
```

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

`bsr pline ;вывод текста`

`endm ;конец макро`

Теперь для вызова нашей подпрограммы можно использовать строку:

`print text ;вывод текста`

Эта строка при ассемблировании заменится на макрос `print text` — это параметр макроса, который подставится на место “?1” в макроопределении. Можно использовать несколько параметров, которые именуются как “?1”, “?2”, “?3”, и т.д..

Можно разрешить (или наоборот, запретить) ассемблеру включать макросы в выходной листинг. Это делается с помощью специальной псевдооперации. Приведем список всех псевдоопераций ассемблера SEKA:

dc определяет переменную или константу в памяти. Размер псевдооперации задается суффиксами `.B`, `.W` или `.L` (по умолчанию используется `.B`). Текстовые константы можно вводить в кавычках или в апострофах, например: `dc.b "Hello",10,13,0`

blk резервирует несколько байт, слов или длинных слов в зависимости от размера псевдооперации. Первый параметр задает число элементов для резервирования, второй определяет шаблон заполнения, например: `blk.w 10,0` — резервирует 10 слов, заполняя их нулями

org задает начальный адрес размещения программы, например, `org $40000`

code задает перемещаемый режим ассемблирования программы

data определяет начало сегмента данных

even добавляет нулевой байт, если адрес нечетный

odd добавляет нулевой байт, если адрес четный

end задает конец текста программы

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

equ,	
=	используется для задания адреса метки, например: Value = 123, или Value equ 123
list	включать текст в листинг (отмена команды nlist). Можно указать следующие параметры:
	с макровыводы
	d макроопределения
	e макрорасширения
	x расширения кода
	Пример: list e ;включать макрорасширения в листинг
nlist	не включать текст в листинг. Список параметров такой же, как и у list
page	задает начало новой страницы (используется при печати на принтер)
if	если параметр этой директивы равен нулю, дальнейший текст ассемблироваться не будет; задает начало блока условного ассемблирования
else	если параметр директивы "if" равен нулю, ассемблирование продолжится здесь (альтернативная ветвь)
endif	определяет конец блока условного ассемблирования
macro	определяет начало макроопределения
endm	определяет конец макроопределения
?n	используется для ссылки на n-й параметр макровывода
?0	сгенерировать уникальное трехзначное число. Используется в макроопределениях, чтобы избежать повторения меток. Например: x?0: bsr pline ;преобразуется в xNNN: bsr pline, где NNN — уникальный номер
illegal	задает несуществующую команду процессора
globl	помечает метку как глобальную. Используется совместно с опцией "I"

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

Итак, после трансляции в памяти находится машинная программа, готовая к выполнению. С помощью запроса "h", Вы можете узнать размер кода программы и ее начальный адрес. Начальный и конечный адреса выводятся в шестнадцатеричной системе счисления, а размер — в десятичной:

Work рабочая область памяти
Src исходный текст программы
RelC таблица смещений кода
RelD таблица смещений данных
Code код программы
Data область данных

Неудобно каждый раз при запуске программы вводить ее адрес, поэтому имеет смысл пометить начало программы в исходном тексте (например, меткой "run:") Тогда для запуска программы можно использовать команду

g run

Запрос "g" (Go) является одним из запросов встроенного отладчика системы SEKA. Приведем весь список запросов отладки:

x	вывести содержимое регистров
xd	вывести и изменить содержимое регистра
gn	начать выполнение программы с адреса n. Система запросит адреса контрольных точек
jn	аналогично "gn", только управление передается по команде JSR. Программа должна завершаться командой RTS
qn	вывести содержимое памяти с адреса n. Можно также указать формат вывода, например q.w \$10000
nn	дизассемблировать область памяти с адреса n
an	начать ввод ассемблерных команд с адреса n
mn	начать редактирование памяти начиная с адреса n. Прервать ввод можно клавишей <Esc>
sn	выполнить n команд, начиная с текущей

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

- f** заполнить область памяти. Все параметры запрашиваются отдельно
- c** выполнить копирование одной области памяти в другую, параметры запрашиваются отдельно
- ?** напечатать значение выражения или адрес метки Например: ? run+\$1000-256
- a** задать последовательность команд для эмуляции запуска программы из CLI
- !** закончить работу с системой SEKA после ввода подтверждения

На примере запроса “?” мы показали, как SEKA может вычислять значения выражений. На самом деле, подобные выражения можно использовать и в исходных текстах программ. Следующие операции обрабатываются ассемблером SEKA:

+	сложение
-	вычитание
*	умножение
/	деление
&	логическое И (AND)
!	логическое ИЛИ (OR)
~	исключающее ИЛИ (XOR)

Также можно задавать различные системы счисления, используя префиксы: “\$” — для шестнадцатеричной, “@” — для восьмеричной и “%” — для двоичной систем. Если префикс опущен, число воспринимается как десятичное.

Но вернемся к отладчику. Как уже было сказано, после ввода команды “g Address” система запрашивает адреса контрольных точек. Можно ввести до 16 адресов, на которых программа должна прерываться для контроля регистров и/или памяти. Если Вы не укажете ни одного контрольного адреса, то предполагается, что Ваша программа завершается директивой ILLEGAL. В противном случае последствия могут быть непредсказуемы.

СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА АССЕМБЛЕРЕ

Отладчик SEKA записывает несуществующую команду (ILLEGAL) в каждую контрольную точку, предварительно сохранив ее первоначальное содержимое. Это приводит к следующему результату: когда процессор встречается команду ILLEGAL, генерируется исключение, вектор которого перехвачен отладчиком. Так что управление передается отладчику, который восстанавливает прежнее содержимое контрольного адреса и выводит строку с информацией о состоянии программы.

Механизм контрольных точек оказывается очень полезным при поиске ошибок в программах. Вы можете, к примеру, поставить контрольную точку в начале подпрограммы, в которой Вы не до конца уверены. После прерывания программы по контрольному адресу, Вы сможете пошагово выполнить "сомнительную" подпрограмму (с помощью запроса "s") и найти возможную ошибку, анализируя строку статуса.

ПЕРВЫЕ ПРОГРАММЫ**Раздел 4. ПЕРВЫЕ ПРОГРАММЫ.**

Итак, мы рассмотрели все основные аспекты программирования на ассемблере. К этому моменту у Вас уже должно быть достаточно знаний, чтобы писать собственные простые программы.

В этом разделе мы рассмотрим некоторые базовые алгоритмы, которые, несомненно, помогут Вам на практике. Хотя все примеры из этого раздела написаны для ассемблера AssemPro, они без труда переносятся на другие системы (см. предыдущий раздел).

Все приводимые ниже алгоритмы оформлены как подпрограммы, так что их можно практически без изменений использовать в других программах. Однако мы все же рекомендуем Вам "поиграть" с этими подпрограммами в отладчике, чтобы лучше понять идеи алгоритмов (в системе SEKA Вы можете использовать команду "j Program_Name" для запуска подпрограммы; результаты обычно возвращаются на регистрах или в памяти).

Начнем с самого простого примера: сложение чисел в таблице.

4.1. Суммирование таблиц (массивов).

Предположим, что Вам нужно получить сумму чисел, записанных в памяти. Пусть у нас имеется пять 16-битных чисел, сумму которых требуется поместить в D0. Вот самый простой способ это сделать:

;(4.1A)

adding1:	clr.l	D0	;обнуление D0 (на всякий случай)
	move	table,D0	;первый элемент — в D0
	add	table+2,D0	;прибавить второй элемент
	add	table+4,D0	;прибавить третий элемент

ПЕРВЫЕ ПРОГРАММЫ

```

      add      table+6,D0    ;прибавить четвертый элемент
      add      table+8,D0    ;прибавить пятый элемент
      rts                      ;возврат
table:  dc.w      2,4,6,8,10
      end

```

Войдите в отладчик, запустите эту программу (если Вы используете ассемблер SEKA, наберите "j adding1") и убедитесь, что в D0 действительно окажется сумма чисел таблицы table.

Приведенный пример работает только с фиксированными адресами. Но вспоминая различные методы адресации процессора мы сможем написать более компактный алгоритм, который к тому же будет способен суммировать разные таблицы.

Для этого мы поместим адрес таблицы в один из адресных регистров (например, в A0) с помощью команды move.l. Тогда, используя этот регистр как базовый указатель, мы сможем применять косвенную адресацию со смещением для доступа к элементам таблицы:

```

;(4.1B)
adding1:  clr.l      D0          ;обнуляем D0
          move.l     #table,A0   ;адрес таблицы — в A0
          move       (A0),D0     ;первый элемент — в D0
          add        2(A0),D0    ;прибавляем второй элемент
          add        4(A0),D0    ;прибавляем третий элемент
          add        6(A0),D0    ;прибавляем четвертый элемент
          add        8(A0),D0    ;прибавляем пятый элемент
          rts                      ;возврат из подпрограммы
table:    dc.w      2,4,6,8,10
          end

```

Оттранслируйте эту программу и войдите в отладчик. Выполняя пошагово все команды до rts, убедитесь, что конечное содержимое D0 совпадает с предыдущим результатом. Заметим, что здесь мы вынуждены использовать команду move.l для записи адреса таблицы в A0, так как адреса в Amiga представляются длинными словами (то же самое можно сделать и с помощью lea: lea table,A0, причем

ПЕРВЫЕ ПРОГРАММЫ

в этом случае суффикс .l не нужен — lea работает только с длинными словами).

А теперь сделаем наш алгоритм еще более компактным, используя "(A0)+" вместо "x(A0)". В этом случае, после каждого обращения к элементам таблицы указатель A0 будет увеличиваться на число считанных байт (в нашем случае на два), так что в конце программы A0 будет указывать на первое неиспользованное слово после таблицы. Можно сделать еще лучше, задавая число суммируемых элементов как отдельную переменную (например, в D1). Для этого нам понадобится организовать цикл: будем после каждого обращения к памяти декрементировать D1, пока не дойдем до нуля. Теперь программа будет выглядеть примерно так:

```

;(4.lC)
adding2:                                ;A0 должен содержать адрес
                                           ;таблицы, а D1 — число элементов
                                           ;обнуляем D0 (быстрее, чем clr.l D0)
loop:                                     ;метка начала цикла
    moveq    #0,D0                       ;прибавляем очередной элемент
    add      (A0)+,D0                    ;уменьшаем счетчик
    subq     #1,D1                       ;повторяем, пока не ноль (можно было
    bne.s    loop                       ;сделать то же самое и с
                                           ;помощью dbra)
                                           ;иначе возврат
    rts
table:      dc.w      2,4,6,8,10
end
    
```

Для тестирования этой программы, запишите в A0 адрес таблицы, а в D1 — число элементов. Затем выполните программу в пошаговом режиме, наблюдая за содержанием регистров (в системе SEKA можно использовать команду "x pc" для установки PC на адрес "adding2", а затем "s" для пошагового выполнения).

Завершая этот пример, попробуйте написать программу для суммирования байтов или длинных слов. Можете также написать программу для вычитания из первого элемента таблицы всех последующих. Например, для табли-

ПЕРВЫЕ ПРОГРАММЫ

цы

table: dc.w 50,8,4,6

у Вас должен получиться результат 50-8-4-6 = 32 (\$20).

4.2. Сортировка таблиц (массивов).

Продолжим работать с таблицами. В этом примере нам понадобится не только читать данные из таблиц, но и изменять их так, чтобы в результате получить отсортированную по возрастанию таблицу.

Для начала выберем алгоритм сортировки. Рассмотрим самый простой метод — метод “пузырька”.

Сначала сравниваем первый элемент со вторым. Если второй элемент больше, то переходим к следующему шагу: сравниваем второй элемент с третьим, и т.д. Если на каждом шаге предыдущий элемент меньше следующего, то таблица уже отсортирована (сортировка не нужна).

Если же на каком-то шаге второй элемент меньше первого, то меняем их местами и устанавливаем специальный флаг. Дойдя до конца таблицы, будем проверять этот флаг: если он установлен, то таблица возможно еще не отсортирована и нужно сделать еще один проход. Иначе — сортировка закончена.

Теперь напомним программу, реализующую этот метод. Для начала определим переменные, которые нам потребуются. Это, во-первых, указатель на таблицу (A0), а также счетчик (D0) и флаг завершения (D1). Так как после каждого прохода сортировки значения A0 и D0 меняются, нам понадобятся переменные для хранения их первоначальных значений (будем использовать A1 и D2). Для организации цикла будем использовать команду DBRA.

Итак, начнем собственно написание программы. Предположим, что перед вызовом подпрограммы сортировки в A1 уже записан адрес таблицы, а в D2 — число элементов.

;(4.2A) заголовочная часть подпрограммы сортировки

sort:	move.l	A1,A0	;дублируем указатель на начало таблицы
	move	D2,D0	;дублируем счетчик элементов

ПЕРВЫЕ ПРОГРАММЫ

```
subq    #2,D0    ;корректируем значение счетчика
moveq   #0,D1    ;очищаем флаг завершения
table:  dc.w     3,6,8,9,5
end
```

На этом подготовка к сортировке закончена: указатель и счетчик установлены, флаг завершения очищен. Теперь поясним смысл команды `subq`. Как мы уже говорили (см. п. 2.4), счетчик для `DBRA` должен быть на единицу меньше, чем число итераций цикла. А так как нам нужно `N-1` итераций для `N` чисел (последний элемент не с чем сравнивать), то корректируем счетчик так, чтобы перед началом цикла его содержимое было `N-2`.

Теперь напомним собственно цикл сравнений. Сравнение двух слов в памяти выглядит примерно так:

```
loop:   move    2(A0),D3    ;следующее слово — в D3
        cmp     (A0),D3    ;сравниваем с текущим
```

Здесь мы вынуждены использовать дополнительный регистр (`D3`), так как `CMP` не допускает адресации обоих операндов в памяти.

Если второй операнд больше или равен первому, то обмен не нужен:

```
bcc.s    poswap    ;ветвление, если больше или равно
```

Иначе нужно поменять местами текущий элемент со следующим (к сожалению, нельзя использовать `EXG`, так как эта команда работает только с регистрами):

```
doswap:  move    (A0),D1    ;сохраняем первый элемент
         move    2(A0),(A0) ;копируем второй элемент
         ;на место первого
         move    D1,2(A0)   ;копируем первый элемент
         ;на место второго
         moveq   #1,D1     ;устанавливаем флаг
```

```
poswap:
```

Теперь инкрементируем указатель и повторяем тело цикла, пока счетчик не станет равным нулю:

```
addq.l   #2,A0    ;счетчик + 2
dbra     D0,loop  ;цикл
```

И наконец, проверяем флаг завершения и повторяем

ПЕРВЫЕ ПРОГРАММЫ

цикл сортировки, если он установлен:

tst	D1	; флаг = 0?
bne.s	sort	; нет — повторить
rts		; да — конец

Вот и все. Приведем полный листинг программы сортировки:

;(4.2B)

run:	move.l	#table,A1	; указатель на таблицу
	moveq	#5,D2	; число элементов
	bsr.s	sort	; вызов подпрограммы сортировки
	illegal		; конец программы
sort:	move.l	A1,A0	; указатель — во временный регистр
	move	D2,D0	; установка счетчика
	subq	#2,D0	; корректировка
	moveq	#0,D1	; очищаем флаг завершения loop:
	move	2(A0),D3	; следующий элемент — в D3
	cmp	(A0),D3	; сравниваем с текущим
	bcc.s	poswap	; ветвление, если больше или равно
doswap:	move	(A0),D1	; сохраняем текущий элемент
	move	2(A0),(A0)	; копируем следующий ; на место текущего
	move	D1,2(A0)	; записываем текущий элемент на место ; следующего
	moveq	#1,D1	; установка флага
poswap:	addq.l	#2,A0	; перемещаем указатель
	dbra	D0,loop	; цикл, пока не обработаем все элементы
	tst	D1	; были перестановки?
	bne.s	sort	; да, повторить
	rts		; иначе возврат
table:	dc.w	10,8,6,4,2	
	end		

Оттранслируйте эту программу, запишите на диск и загрузите в отладчик. Запустите программу с адреса "run:" и выберите пункт "Parameter-Display-Disassembled". Если Вы правильно ввели программу, то элементы таблицы table теперь будут следовать в порядке возрастания (2, 4, 6, 8, 10). Написанная нами подпрограмма "портит" некоторые регистры, что обычно весьма нежелательно (ведь вызывающая программа может использовать эти регистры для своих це-

ПЕРВЫЕ ПРОГРАММЫ

лей). Поэтому в начале любой подпрограммы целесообразно сохранять регистры (например, в стеке), а перед возвратом — восстанавливать. Для этого удобно использовать команду MOVEM (см. п.2.4):

```
sort:    movem.l D0-D7/A0-A6,-(SP)    ;сохраняем регистры
loop1:
    ...                                ;наша программа сортировки
    ...
    tst      D1                        ;... проверка флага
    bne.s    loop1                    ;вместо bne sort
    movem.l  (SP)+,D0-D7/A0-A6        ;восстанавливаем регистры
    rts                                           ;все!
```

Прежде чем закончить пример с сортировкой, попробуйте самостоятельно написать программу, сортирующую таблицы в порядке убывания.

То, что в качестве примера приведена сортировка методом “пузырька”, вовсе не означает, что этот метод нужно использовать на практике. На самом деле этот метод является крайне медленным и неэффективным, но он прекрасно подходит для ознакомления с некоторыми нюансами программирования на ассемблере. Напомним, что этот раздел книги является вводным, а понять основы машинного программирования на примере алгоритма быстрой сортировки (quick sort) было бы, согласитесь, намного труднее. Сказанное относится и к другим примерам, приведенным в этой книге.

4.3. Перевод систем счисления.

Как мы уже упоминали в п.1.2.3, перевод чисел из одной системы счисления в другую очень часто используется в машинном программировании. Числа, которые вводятся с клавиатуры или выводятся на экран, представляются в виде строк символов. Чтобы обрабатывать такие числа, нужно уметь переводить их из символического представления в двоичное и обратно. Такой перевод тесно связан с преобразованиями систем счисления.

Далее мы рассмотрим основные алгоритмы таких

ПЕРВЫЕ ПРОГРАММЫ

преобразований и приведем соответствующие примеры. Начнем с наиболее простого алгоритма.

4.3.1 Перевод шестнадцатеричных чисел в ASCII-представление.

Для начала определим начальные и конечные данные алгоритма. Пусть регистр D1 содержит 32-битное число, которое должно быть переведено в ASCII-строку длиной 8 символов (результат будем записывать в память).

Как мы знаем, для перевода двоичного числа в шестнадцатеричное нужно разбить его на группы по четыре бита. Каждая из полученных групп будет содержать шестнадцатеричную "цифру". Пусть регистр D2 содержит одну из таких групп, и нам требуется получить ASCII-код соответствующей цифры. Для этого будем использовать следующую таблицу:

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
\$30	\$31	\$32	\$33	\$34	\$35	\$36	\$37	\$38	\$39	\$41	\$42	\$43	\$44	\$45	\$46

В этой таблице показаны ASCII-коды шестнадцатеричных цифр. Легко видеть, что если D2 содержит число из диапазона 0-9, то для преобразования его в ASCII-код к D2 нужно прибавить число \$30. Но для преобразования остальных чисел (10-15) нужно прибавлять не \$30, а \$37. Поэтому нам придется разделить эти два случая в нашем алгоритме.

Теперь мы готовы написать фрагмент программы для преобразования D2 в цифру:

```
nibble:  and  #S0F,D2      ;рассматриваем только младший полубайт
          add  #$30,D2     ;прибавляем $30
          cmp  #$3A,D2     ;преобразуем цифру?
          bcs.s ok         ;да, уже преобразовали
          addq #7,D2       ;иначе преобразуем букву — добавляем
                               ;еще 7
ok:      rts              ;все
```

Эта программа возвращает в D2 ASCII-код полубайта. Чтобы преобразовать целый байт, нужно вызвать эту

ПЕРВЫЕ ПРОГРАММЫ

подпрограмму дважды:

;(4.3.1A) bin-hex

```

;
    lea    buffer,A0    ;начало Вашей программы
    moveq  #$4A,D1      ;буфер для записи результата
    bsr.s  byte         ;байт, который надо преобразовать
    rts                    ;вызов конвертора

;
    ...                ;продолжение Вашей программы byte:
    move   D1,D2        ;исходный байт — в D2
    lsr    #4,D2        ;передвигаем старший полубайт на место
                        ;младшего
    bsr.s  nibble       ;преобразовать полубайт
    move.b D2,(A0)+      ;записать первую цифру результата
    move   D1,D2        ;исходный байт — в D2
    bsr.s  nibble       ;преобразовать младший полубайт
    move.b D2,(A0)+      ;записать вторую цифру результата
    rts                    ;возврат

nibble:
    and    #$0F,D2      ;оставляем только младший полубайт
    add    #$30,D2      ;прибавляем $30
    cmp    #$3A,D2      ;цифра?
    bcs.s  ok           ;да, в D2 — код
    addq   #7,D2        ;иначе добавляем 7 (чтобы в сумме
                        ;получить $37)

ok:
    rts                    ;возврат

buffer:
    blk.b  9,0          ;место для записи результата
    end
    
```

Чтобы протестировать эту программу, оттранслируйте ее и загрузите в отладчик. Затем поставьте контрольную точку на первую команду `rts` (для этого нужно нажать <right-Amiga> , предварительно указав мышью на нужную строку). Выполните программу, наблюдая за содержимым регистра D2. После этого выберите пункт меню "Parameter-Display-HEX-Dump" и убедитесь, что буфер действительно содержит строку "4A".

Обратите внимание на очередность обработки полубайтов: сначала мы конвертируем старший полубайт, а за-

ПЕРВЫЕ ПРОГРАММЫ

тем младший (так как строки выводятся на экран слева-направо). При этом мы выделяем буфер такого размера, чтобы туда поместилось ASCII-представление длинного слова (8 символов), а за строкой символов всегда следовал нулевой байт (это нужно для правильной работы подпрограммы вывода строк на экран, но об этом позже). Теперь осталось написать программу для преобразования длинных слов.

В случае длинных слов нам нужно обращать особое внимание на порядок обработки полубайтов. Перед первым вызовом подпрограммы `nibble` самый старший полубайт должен быть помещен на первое место, при этом никакие биты не должны теряться.

Понятно, что команда `LSR` для этого не совсем подходит. Поэтому будем использовать команды прокрутки `ROR` и `ROL` (напомним, что эти команды гарантируют сохранение всех битов операнда).

Если мы прокрутим содержимое `D1` влево на четыре бита, то старший полубайт окажется на месте младшего и можно будет вызвать подпрограмму `nibble`. Повторив те же действия еще семь раз, мы получим нужный результат, причем регистр `D1` вернется к своему первоначальному значению:

```
;(4.3.1B) bin-hex-2
hexlong:
```

<code>lea</code>	<code>buffer,A0</code>	;адрес буфера — в <code>A0</code>
<code>move.l</code>	<code>#\$12345678,D1</code>	;число для преобразования
<code>moveq</code>	<code>#8-1,D3</code>	;счетчик полубайтов <code>loop</code> :
<code>rol</code>	<code>#4,D1</code>	;старший полубайт — в начало
<code>move</code>	<code>D1,D2</code>	;данные для <code>nibble</code>
<code>bsr.s</code>	<code>nibble</code>	;конвертируем полубайт
<code>move.b</code>	<code>D2,(A0)+</code>	;результат — в буфер
<code>dbra</code>	<code>D3,loop</code>	;повторить 8 раз
<code>rts</code>		;и все!

`nibble:`

<code>and</code>	<code>#\$0F,D2</code>	;оставляем младший полубайт
<code>add</code>	<code>#\$30,D2</code>	;прибавляем <code>\$30</code>
<code>cmp</code>	<code>#\$3A,D2</code>	;цифра?

ПЕРВЫЕ ПРОГРАММЫ

```
bcs.s    ok           ;да, код — в D2
addq     #7,D2        ;иначе добавляем еще 7 ok: rts ;все
buffer:
blk.b     9,0         ;место для результата плюс один
                    ;резервный нулевой байт
end
```

Оттранслируйте эту программу и загрузите в отладчик, затем поставьте контрольную точку на первую команду возврата (rts) и запустите. Просматривая результат, Вы обнаружите, что программа сработала неправильно — вместо строки "12345678" выходной буфер будет содержать "56785678". Попробуйте найти ошибку!

А ошибка здесь довольно типичная: в команде прокрутки не указан суффикс .L. Ассемблер воспринимает размер команды "rol" как .W (по умолчанию), и в результате сдвигалось только младшее слово. Поэтому мы получили одни и те же значения дважды. Замените "rol" на "rol.l" и программа будет работать правильно.

Эта ошибка наводит на мысль о том, как легко можно переделать эту программу для конвертирования 16-битных чисел: нужно просто опустить суффикс ".l" в команде прокрутки и поменять начальное значение счетчика цикла с 7 на 3. Вот и все.

И в завершении попробуйте изменить программу так, чтобы она переводила шестнадцатеричные числа в ASCII-строки из шести цифр.

Теперь рассмотрим задачу посложнее, а именно — перевод четырехразрядных десятичных чисел (0-9999) в ASCII-представление.

4.3.2. Перевод десятичных чисел в ASCII-представление.

В случае десятичных чисел механизм разбиения на битовые группы не работает, поэтому нам понадобится другой метод.

Рассмотрим подробнее структуру десятичных чисел. В четырехразрядном числе самый старший разряд задает тысячи, следующий — сотни, и т.д.

ПЕРВЫЕ ПРОГРАММЫ

Отсюда видно, что если Вы разделите исходное число на 1000, то в результате как раз получится значение старшего разряда этого числа. Если Вы теперь разделите остаток на 100, то получите уже разряд сотен, а разделив полученный остаток на 10 — разряд десятков. В качестве значения разряда единиц нужно просто взять остаток последнего деления (на 10).

Реализующая этот метод программа будет выглядеть

так:

```
main:      lea      buffer,A0      ;адрес буфера — в A0
           move    #1234,D1       ;исходное десятичное число
           bsr.s   deci_4         ;вызов конвертора
           illegal          ;место контрольной точки
deci_4:    ;подпрограмма-конвертор
           divu    #1000,D1       ;делим на 1000
           bsr.s   digit         ;обработка результата и пересылка
                                   ;остатка
           divu    #10,D1        ;делим на 10
           bsr.s   digit         ;обработка результата и пересылка
                                   ;остатка
                                   ;обработка остатка:
digit:     add     #$30,D1        ;переводим результат в ASCII
           move.b  D1,(A0)+       ;записываем в цифру в буфер
           clr     D1             ;очищаем младшее слово (moveq #0
                                   ;использовать нельзя, так как в этом
                                   ;случае обнулится весь регистр)
           swap    D1             ;остаток — в младшее слово
           rts     ;возврат
buffer:    blk.b   5,0           ;место для записи результата
           end
```

Здесь мы использовали небольшой трюк, который является типичным для ассемблерного программирования: после трехкратного вызова digit мы попадаем прямо в эту подпрограмму, не используя при этом команды BSR или JSR. Как только управление доходит до команды rts, происходит возврат в программу main, а не в deci_4. Таким образом мы имеем неявный четвертый вызов подпрограммы

ПЕРВЫЕ ПРОГРАММЫ

digit и возврат в главную программу.

Протестируйте эту программу с помощью отладчика для разных начальных значений (помните, что эта программа корректно работает только с числами из диапазона 0-9999).

Теперь переходим к рассмотрению обратной задачи — задачи преобразования символьных строк в двоичные числа.

4.3.3. Перевод ASCII-строк в шестнадцатичные числа.

Как мы уже говорили, каждый символ ASCII-записи шестнадцатичного числа задает его отдельный полубайт. В п. 4.3.1 мы уже использовали этот факт, однако теперь нам нужно написать в некотором смысле обратную программу.

У нас есть две альтернативы: количество шестнадцатичных цифр

1. известно заранее
2. неизвестно

Если считать количество цифр фиксированным, то программу написать проще, однако здесь есть один недостаток: некоторые числа придется дополнять слева нулями (например, число 1 нужно будет вводить как "0001"). Это нас вряд-ли устроит, поэтому будем исходить из второй альтернативы.

Сначала напишем программу для перевода одного символа. Пусть адрес символа находится в A0, а результат требуется получить в D0. Имеем:

start:

move.l	#string,A0	;адрес символа
bsr.s	nibblein	;получить значение
illegal		;место для контрольной точки
nibblein:		;получить число по (A0)
moveq	#0,D0	;очищаем D0 (т.к следующая команда изменяет только младший байт)
move.b	(A0)+,D0	;извлекаем код цифры
sub	#'A',D0	;вычитаем \$41
bcc.s	ischar	;ветвление, если была буква (A-F)

ПЕРВЫЕ ПРОГРАММЫ

```
ischar:    addq    #7,D0          ;иначе была цифра (0-9) - корректируем
           add     #$A,D0        ;корректируем окончательно (ADDQ
                                   ;использовать нельзя: $A > 8)
           rts
string:    dc.b    'B',0        ;исходный символ ('B', код $42)
           end
```

Рассмотрим, как работает эта программа. Первая команда устанавливает регистр A0 на адрес символа 'B', после чего вызывается подпрограмма- конвертор (nibblein).

Внутри конвертора в D0 записывается ASCII-код цифры (в нашем случае \$42), из которого затем вычитается число \$41. В результате D0 будет содержать 1 и ветвление сработает, а после прибавления \$A получим окончательный результат — число \$B.

А если бы буфер содержал цифру (0-9)? Тогда результат первого вычитания был бы отрицательным и ветвление не выполнилось бы. Возьмем, например, цифру 5: ASCII-код символа '5' — это \$35. После вычитания \$41 имеем -12 и флаг C установлен. Поэтому к D0 прибавится число 7, а затем и \$A (10). В результате получим 5.

У этой подпрограммы есть один недостаток, а именно — отсутствие контроля за правильностью входных данных: если в буфер записать любой символ, не являющийся шестнадцатеричной "цифрой", то в D0 может оказаться все, что угодно. Однако здесь мы для краткости опустим проверку на допустимые значения, при необходимости Вы можете добавить ее сами.

Переходим к написанию конвертора многосимвольных шестнадцатеричных чисел.

Для начала заметим, что самая первая цифра представляет старший полубайт числа. Чтобы учесть это, а также то, что нам неизвестно количество символов во входной строке (на самом деле будем предполагать, что результат поместится в 32-битное слово), используем небольшой трюк. Сначала приведем текст программы (перед вызовом под-

ПЕРВЫЕ ПРОГРАММЫ

программы-конвертора A0 должен содержать адрес входной строки; результат возвращается в D1):

```

hexin:      ;тестовая программа
            ;для начала обнуляем D1
            moveq    #0,D1
            lea      string(PC),A0 ;адрес строки - в A0 (lea через PC -
            ;эффективнее и короче, чем move.l)
            bsr.s    hexinloop     ;вызов подпрограммы-конвертора
            illegal   ;для отладки

hexinloop:
            tst.b    (A0)           ;конец строки? (нулевой байт)
            beq.s    hexinok       ;да, выход
            bsr.s    nibblein      ;иначе продолжаем: вызов конвертора
            ;текущей цифры
            lsl.l    #4,D1         ;сдвигаем результат
            or.b     D0,D1         ;"вставляем" полубайт
            bra.s    hexinloop     ;цикл hexinok:
            rts         ;возврат

nibblein:
            ;конвертор ASCII-цифры в полубайт
            moveq    #0,D0         ;очищаем D0 от "мусора"
            move.b   (A0)+,D0      ;извлекаем текущую цифру,
            ;перемещаем указатель
            sub      #'A',D0       ;вычитаем $41
            bcc.s    ischar        ;символ из диапазона 'A'-'F'
            addq     #7,D0         ;иначе корректируем ischar:
            add      #'A',D0       ;корректируем окончательно
            rts         ;все

string:     dc.b     "56789ABC",0 ;ASCII-запись исходного числа плюс
            ;нулевой байт как признак конца строки

end
    
```

Протестируйте эту программу с помощью отладчика, наблюдая за содержимым регистра D1.

А трюк состоит в том, что мы каждый раз сдвигаем результат на 4 бита, освобождая тем самым место для следующего полубайта. Таким образом, самый последний полубайт окажется в младшей позиции, и D1 будет содержать правильный результат.

Чтобы обеспечить контроль входных данных, нам потребуется немного изменить программу, добавив после вы-

ПЕРВЫЕ ПРОГРАММЫ

зова nibblein проверку корректности символов. А именно, если содержимое D0 после возврата из nibblein больше чем \$F, то текущий символ не является цифрой (0-9, A-F) и нужно закончить обработку строки. Такую проверку можно задать несколькими способами, мы будем использовать самый простой из них — cmp №\$10,D0: если флаг C очищен, то содержимое D0 больше или равно \$10.

Такое изменение программы, кстати, позволит нам опустить проверку конца строки — ведь нулевой байт также не является ASCII-кодом цифры. Приведем текст программы с контролем входных данных:

;(4.3.1C) hex-conv2

```
hexin:                                ;тестовая программа
      moveq    #0,D1                  ;обнуляем D1
      lea      string(PC),A0         ;адрес строки — в A0
      bsr.s    hexinloop              ;вызов конвертора
      illegal   ;для отладки

hexinloop:
      bsr.s    nibblein               ;конвертируем текущий символ
      cmp      #$10,D0                ;проверка на правильность
      bcc.s    hexinok                ;текущий символ — не из диапазона
                                      ;0-9,A-F
      lsl.l    #4,D1                  ;сдвигаем результат
      or.b     D0,D1                  ;накладываем следующий полубайт
      bra.s    hexinloop

hexinok:
      rts                                     ;возврат

nibblein:
      moveq    #0,D0                  ;очищаем D0 от "мусора"
      move.b   (A0)+,D0                ;извлекаем очередную цифру,
                                      ;перемещаем указатель
      sub      #'A',D0                ;вычитаем $41
      bcc.s    ischar                 ;символ из диапазона 'A'-'F'
      addq     #7,D0                  ;иначе корректируем

ischar:
      add      #'A',D0                ;корректируем окончательно
      rts                                     ;все

string:
      dc.b     "56789ABC",0           ;ASCII-запись исходного числа плюс
                                      ;нулевой байт как признак конца строки

      end
```

ПЕРВЫЕ ПРОГРАММЫ

Итак, мы рассмотрели алгоритм преобразования шестнадцатиричных ASCII-чисел в двоичные. Немногим сложнее оказывается и алгоритм для перевода чисел, записанных в десятичной системе.

4.3.4. Преобразование ASCII-строк в десятичные числа.

Здесь мы будем использовать метод, аналогичный предыдущему. Единственное отличие состоит в том, что для сдвига десятичного числа на один разряд влево нам придется использовать команду умножения на 10 (напомним, что в предыдущем примере мы обошлись командой lsl). Программа будет выглядеть так:

```

:(4.3.4) decimal-conv
decin:                                     ;тест конвертора десятичных чисел
      moveq    #0,D1                      ;обнуляем D1
      lea      string(PC),A0              ;загружаем указатель строки
      bsr.s    decinloop                  ;вызов конвертора
      illegal   ;для отладки
decinloop:                                ;подпрограмма конвертирования
                                           ;десятичных чисел
      bsr.s    digitin                    ;чтение цифры
      cmp      #10,D0                     ;проверка на корректность
      bcc.s    decinok                    ;ошибка — конец обработки
      mulu     #10,D1                      ;смешаем результат на один разряд
      add      D0,D1                      ;добавляем следующую цифру
      bra.s    decinloop                  ;цикл decinok:
      rts                                             ;возврат
digitin:                                ;преобразование кода цифры
      moveq    #0,D0                      ;очищаем весь D0
      move.b    (A0)+,D0                  ;извлекаем очередную цифру,
                                           ;перемешаем указатель
      sub      #'0',D0                    ;вычитаем код символа '0'
      rts                                             ;все
string:  dc.b    '123456',0               ;ASCII-запись десятичного числа для
                                           ;конвертирования
end

```

Загрузите эту программу в отладчик и выполните в пошаговом режиме, наблюдая за содержимым регистра D1.

Обратите внимание на то, что эта программа может

ПЕРВЫЕ ПРОГРАММЫ

работать только с числами не больше 655350. Это связано с тем, что команда MULU может умножать только 16-битные слова, поэтому наибольший результат, который можно получить с помощью MULU, определяется как $\$FFFF * 10 = 65535 * 10 = 655350$. Однако это не слишком серьезное ограничение и мы не будем переписывать программу для обработки больших чисел.

Раздел 5. ВНЕШНИЕ РЕГИСТРЫ.

Как мы уже говорили, процессор Amiga может адресовать не только ячейки оперативной и постоянной памяти, но и специальные регистры — регистры внешних устройств. Именно регистры внешних устройств обеспечивают интерфейс обмена информацией между процессором и остальным “железом” (hardware).

Для каждой компоненты hardware выделяется один или несколько регистров, каждый из которых имеет свой адрес. Обращение процессора к этим компонентам происходит как обычное обращение к памяти по этим адресам (с помощью уже известных Вам команд MOVE, AND, OR итд.). Нетрудно понять, что на языке ассемблера можно писать программы, использующие все возможности Amiga.

В этом разделе мы рассмотрим основные функции custom-чипов и других устройств Amiga и на примерах покажем, как эти функции можно использовать в программах.

5.1. Работа со специальными клавишами.

Загрузите AssemPro, войдите в отладчик, выберите пункт меню “Parameter- Display-From-Address” и введите адрес \$BFEC00. Затем выберите пункт “Parameter-Display-HEX-Dump” (в системе SEKA наберите “q \$bfec00”).

Вы увидите список адресов начиная с \$BFEC00, в котором два байта всегда повторяются. Эти два байта отражают состояния двух внешних регистров.

Повторение одних и тех же байтов по разным адресам связано с тем, что при декодировании адреса используются не все биты. В нашем примере для адресации регистров используются только два старших байта и младший бит (нулевой). Адреса таких регистров записываются

ВНЕШНИЕ РЕГИСТРЫ

как `$BFECxx`, где байт `xx` не несет никакой информации о нужном адресе в битах 1-7, и только значение бита 0 определяет, какой из двух регистров используется. Такая на первый взгляд странная форма адресации используется для доступа к большинству внешних регистров Amiga.

Посмотрим, какая информация содержится в регистрах `$BFECxx`. Нажмите клавишу `<Alt>` и выберите "Parameter-Display-HEX-Dump" (в системе SEKA наберите "`q $bfec00`" и нажмите `<Alt>` сразу после `<Enter>`). Вы увидите, что содержимое регистра `$BFEC01` изменилось и стало равно `$37`. `$BFEC01` — это регистр статуса специальных клавиш. Следующая таблица показывает влияние клавиш на его содержимое:

левый Shift	<code>\$3F</code>
правый Shift	<code>\$3D</code>
Ctrl	<code>\$39</code>
Alt	<code>\$37</code>
левая Amiga	<code>\$33</code>
правая Amiga	<code>\$31</code>

Вы можете использовать этот регистр для проверки нажатия указанных клавиш, например:

```
skeys = $bfec01
...
cmp.b  #$37,skeys ;клавиша Alt нажата?
beq    function   ;да! вызвать function
cmp.b  #$31,skeys ;а правая Amiga?
beq    function2  ;да! вызвать function2
... ;и так далее...
```

5.2. Работа с таймером.

Если Вам требуется узнать, сколько времени прошло между двумя событиями, используйте встроенный таймер, который обеспечивает максимальную точность и работает независимо от процессора (на самом деле Amiga имеет несколько таймеров, но здесь мы рассмотрим только один из них — 24-битный CIA-A TOD таймер).

Таймеры (или счетчики) — это специальные устрой-

ВНЕШНИЕ РЕГИСТРЫ

ства, которые периодически уменьшают или увеличивают значения своих регистров. Рассматриваемый нами таймер входит в состав CIA (Complex Interface Adapter — набор чипов, обеспечивающих ввод/вывод и другие интерфейсные функции) и работает со скоростью 60 герц (или 50 в режиме PAL). Это означает, что значение его регистра увеличивается на единицу 60 раз в секунду.

Регистр нашего таймера разбит на три байта, которые имеют адреса \$BFF801, \$BFF901 и \$BFFA01. В связи с этим его значение не может быть прочитано одной командой MOVE.L.

Приведем пример программы, использующей таймер для измерения времени работы некоторой подпрограммы:

```
test:      bsr.s    gettime      ;получить текущее значение счетчика
                                   ;таймера
            move.l   D7,D6      ;сохранить результат в D6
            bsr.s    routine     ;вызов тестовой подпрограммы
            bsr.s    gettime     ;снова получить значение счетчика
            sub      D6,D7      ;вычислить время работы подпрограммы
                                   ;routine в 1/60 долях секунды
            illegal   ;для отладки
routine:    move     #500,D0     ;счетчик задержки loop:
            dbra     D0,loop     ;задержка
            rts       ;и возврат
gettime:    ;получение текущего значения таймера
            move.b   $BFEA01,D7 ;старший байт — в D7
            lsl.l    #8,D7      ;сдвиг D7 на 1 байт влево
            move.b   $BFE901,D7 ;добавляем средний байт
            lsl.l    #8,D7      ;опять сдвигаем
            move.b   $BFE801,D7 ;последний (младший) байт счетчика
            rts       ;все
```

На самом деле область применения таймеров Amiga гораздо шире (например, операционная система использует таймеры для организации многозадачности и для других служебных целей). Таймеры могут вызывать прерывание программы по определенным событиям (например при переполнении счетчика), что позволяет синхронизировать

ВНЕШНИЕ РЕГИСТРЫ

различные действия по времени. Информацию о том, как программировать таймерные прерывания, можно найти в технической документации Amiga.

5.3. Работа с мышью и джойстиком.

Amiga имеет два регистра, в которых отображается статус мыши и джойстика. Как ни странно, оба эти устройства используют один и тот же порт, хотя принципы их работы сильно различаются.

Джойстик генерирует стабильный сигнал на выходе, соответствующем его положению, в то время как мышь выдает множество коротких импульсов — два при горизонтальном и два при вертикальном перемещении.

Ясно, что компьютер должен постоянно следить за состоянием этих портов, чтобы своевременно получать сигналы от мыши и вычислять новое положение указателя. К счастью, этим занимаются custom-чипы, освобождая процессор для более важной работы.

Итак, Вы можете получить текущее состояние мыши/джойстика из регистров \$DFF00A и \$DFF00C (для портов 1 и 2 соответственно). Старшие байты этих регистров содержат информацию о вертикальном перемещении мыши, а младшие — о горизонтальном.

Но будьте внимательны — не пытайтесь выводить содержимое этих регистров с помощью AssemPro, так как это приводит к зависанию компьютера. Выглядит это довольно занимательно (экран начинает "мелькать"), однако выйти из этого состояния можно только по RESETу, в результате чего вся информация в памяти компьютера теряется.

Напишем небольшую программу для чтения этих регистров:

```
;(5.3A) mouse
test:      bsr.s run           ;тестовая подпрограмма
           bra.s test         ;"вечный" цикл
           illegal
joy = $DFF00A
```

ВНЕШНИЕ РЕГИСТРЫ

run:

```

    move joy,D6      ;данные порта 1 в D6
    move joy+2,D7    ;данные порта 2 в D7
    bra.s run        ;rts в случае SEKA
end

```

Если теперь оттранслировать эту программу и запустить в режиме "Breakable", то содержимое регистров D6 и D7 будет периодически обновляться на экране и Вы сможете наблюдать за состоянием портов мыши/джойстика.

Передвиньте мышь и Вы увидите, что содержимое регистра D6 изменится. При горизонтальном движении мыши меняется только младший байт, а при вертикальном — старший.

Нетрудно убедиться, что содержимое порта мыши зависит от направления и скорости перемещения мыши, однако оно не определяет текущие координаты указателя (переместите указатель в левый верхний угол и продолжайте двигать мышь влево — содержимое D6 все равно будет меняться). Модифицируем нашу программу чтения регистров так, чтобы она выдавала разность предыдущего и текущего значений:

```
;(5.3B)                                ;программа чтения данных мыши
```

test:

```

    bsr.s run        ;тестовая подпрограмма
    bra.s test       ;"вечный" цикл
    illegal

```

joy = \$DFF00A

run:

```

    move D7,D6      ;копируем старое содержимое в D6
    move joy,D7     ;новое содержимое
    sub D7,D6       ;разность в D6
    bra.s run       ;rts в случае SEKA
end

```

Запустите эту программу и понаблюдайте за содержимым D6. Вы увидите, что D6 содержит величину перемещения мыши с момента последней проверки (старший байт содержит перемещение по вертикали, младший — по гори-

ВНЕШНИЕ РЕГИСТРЫ

зонтали). Можно отслеживать и абсолютное положение указателя мыши — надо просто периодически проверять порты и прибавлять текущее перемещение к его координатам.

Теперь рассмотрим влияние джойстика на значения регистров портов. Предположим, что у Вас есть джойстик, присоединенный ко второму порту. Измените адрес `joу` на `$DFF00C` и запустите предыдущую программу в режиме "Breakable". Наблюдая за D6 Вы увидите, что этот регистр содержит либо ноль, либо копию D7 (при использовании SEKA Вам придется запустить программу два раза).

Переведите джойстик в положение "вверх" — со-держимое D6 будет теперь равно `$FF00`. Если Вы отпусти-те джойстик, то значение D6 изменится и станет равным `$100`. Это означает, что при "движении" вверх из старшего байта порта вычитается единица, а при возвращении джой-стика в исходное положение — прибавляется То же самое происходит и при "движении" влево, только в этом случае изменяется младший байт порта.

В следующей таблице показано изменение значений регистра порта при различных положениях джойстика:

вверх	<code>\$FF00</code>	старший байт минус 1
вниз	<code>\$00FF</code>	младший байт минус 1
влево	<code>\$0100</code>	старший байт плюс 1
вправо	<code>\$0001</code>	младший байт плюс 1

Эти значения тем не менее не являются абсолютно надежными если Вы достаточно быстро "прокрутите" джойстик, то регистр D6 будет содержать бессмыслицу (это происходит из-за того, что контроллер начинает восприни-мать джойстик как мышь). Несмотря на это, описанный спо-соб получения сигналов от джойстика является наиболее быстрым.

Теперь нам осталось выяснить, как получать инфор-мацию о кнопках мыши или джойстика. Для этого сущест-вуют два специальных бита в регистре `$BFE001`: бит 6 от-

ВНЕШНИЕ РЕГИСТРЫ

ражает состояние левой кнопки мыши (порт 1), а бит 7 — джойстика (порт 2): если один из этих битов установлен, то соответствующая кнопка не нажата.

Например, для проверки нажатия кнопки джойстика можно использовать следующий фрагмент программы:

```
tst.b $BFE001    ;кнопка нажата?
bpl fire        ;да!
```

Здесь команда `tst.b` служит для проверки знакового бита регистра `$BFE001`, а так как знаковый бит (бит 7) как раз является флагом состояния кнопки джойстика, то следующая команда ветвления сработает, если кнопка была нажата (так как в этом случае бит 7 равен нулю и содержимое `$BFE001` положительно).

Вот пример законченной программы для проверки состояния джойстика:

```
;(5,3C) fire button & joy difference
test:
    bsr.s    run        ;тестовая подпрограмма
    tst.b    $BFE001    ;проверка нажатия кнопки
    bpl.s    fire       ;ветвление, если кнопка нажата
    bsr.s    test       ;"вечный" цикл
joy = $DFF00C
run:
    move     D7,D6      ;старое содержимое порта в D6
    move     joy,D7     ;новое содержимое в D7
    sub      D7,D6      ;разность в D6
    bsr.s    run        ;rts в случае SEKA
fire:
    nop
end
```

5.4. Работа со звуком.

Amiga имеет достаточно развитое hardware для генерации звуков. В состав библиотек операционной системы входит множество подпрограмм для работы со звуком, однако они представляют интерес в основном для программистов на C или BASIC'е, мы же обсудим программирование звука на самом низком уровне — на уровне hardware.

ВНЕШНИЕ РЕГИСТРЫ

Все звуковые функции в Amiga обеспечиваются чипом Paula, доступ к которому осуществляется через специальные регистры. Ясно, что никакой язык программирования высокого уровня и никакая библиотека функций не могут предоставить Вам больше возможностей, чем непосредственное программирование чипа Paula.

Как же Amiga воспроизводит звук? Paula использует механизм DMA (Direct Memory Access — прямой доступ к памяти) для доступа к звуковым данным. Так что Вам нужно всего-лишь указать место в памяти, где записаны параметры тона (или любого другого звука). В дополнении к этому нужно передать чипу Paula информацию о том, как эти параметры должны интерпретироваться.

Начнем с самого простого — с генерации тона постоянной частоты. Если нарисовать диаграмму такого звука, то она будет состоять из последовательности повторяющихся фрагментов, форма которых определяет так называемый “волновой пакет” тона. Существует несколько стандартных волновых пакетов: синусоидальный, прямоугольный, треугольный и пилообразный. Рассмотрим для начала самый простой из них — прямоугольный.

Очевидно, что для генерации “прямоугольного” тона нужно периодически переключать динамик из одного положения в другое. Частота таких переключений и определяет частоту тона.

Чтобы воспроизвести любой звук с помощью Amiga, нужно задать таблицу всех значений его амплитуды (каждый элемент таблицы должен лежать в пределах от -128 до 127). В случае простого прямоугольного тона нам потребуется таблица, содержащая всего два значения амплитуды — наибольшее и наименьшее, например:

```
soundtab: dc.b -100,100
```

Теперь нужно передать адрес этой таблицы чипу Paula. Так как Amiga имеет четыре звуковых канала, мы можем записать адрес таблицы амплитуд в любой из следующих регистров:

ВНЕШНИЕ РЕГИСТРЫ

\$DFF0A0 для канала 0 (левый динамик)
\$DFF0B0 для канала 1 (правый динамик)
\$DFF0C0 для канала 3 (левый динамик)
\$DFF0D0 для канала 4 (правый динамик)

Выберем, например, нулевой канал:

`move.l #soundtab,$DFF0A0` ;адрес таблицы амплитуд

Далее нам нужно проинформировать звуковой чип о том, сколько элементов содержит таблица амплитуд. При воспроизведении звука данные из этой таблицы выбираются последовательно, причем при достижении конца таблицы звук “зацикливается”: указатель DMA сбрасывается на начало таблицы. Поскольку custom-чипы адресуют память по словам (16 бит), а не по байтам, то таблица амплитуд должна содержать четное число байт, а ее длина должна задаваться в словах. Для прямоугольного тона мы должны записать число 1 (длина нашей таблицы — одно слово) в регистр **\$DFF0A4**:

`move #1,$DFF0A4` ;длина таблицы амплитуд в словах

Для остальных каналов адреса “звуковых” регистров получаются добавлением $x * \$10$, где x — номер канала.

После этого мы должны определить, с какой скоростью данные должны извлекаться из памяти и выводиться на динамик. Эта скорость будет задавать частоту воспроизводимого тона. Однако в Amiga частота задается как бы наоборот: чем больше значение регистра частоты, тем ниже сама частота (фактически, задается период, умноженный на некоторый коэффициент). Выберем, для примера, значение 600:

`move #600,$DFF0A6` ;скорость DMA

И наконец зададим громкость генерируемого тона (значение громкости должно лежать в диапазоне 0-65). Выберем, например, громкость 40:

`move #40,$DFF0A8` ;уровень громкости

Мы предоставили чипу Paula всю информацию, однако никакого звука пока не слышно. Последнее, что нам нужно сделать, это запустить собственно DMA. Управлять

ВНЕШНИЕ РЕГИСТРЫ

каналами DMA можно с помощью специального регистра, адрес которого равен \$DFF096. Нам понадобятся только шесть бит этого регистра:

- | | |
|-----------------|--|
| Бит 15 (\$8000) | Если этот бит равен 1, то биты регистра, соответствующие установленным битам записываемых значений, будут устанавливаться, иначе эти биты будут очищаться. Те биты регистра, которым соответствуют нулевые биты записываемого слова, изменяться не будут. Такой механизм обеспечивает дополнительную безопасность при работе с регистром, так как он также содержит информацию о дисковых операциях, которая не должна изменяться пользователем. |
| Бит 9 (\$200) | Этот бит определяет, может ли Paula использовать DMA. Для воспроизведения тона этот бит должен быть установлен. |
| Биты 0-3 | Включают/выключают соответствующий звуковой канал (1 — канал включен). |

Для того, чтобы включить DMA для воспроизведения нашего тона, нам нужно написать:

```
move    #8000+$200+1,$DFF096    ;запуск DMA
```

Теперь приведем пример программы, воспроизводящей синусоидальный тон:

```

** Sound generation using hardware registers ** (5.5A)
ctlw = $DFF096                ;регистр управления DMA
c0thi = $DFF0A0                ;старшее слово регистра адреса таблицы
c0tlo = c0thi + 2              ;младшее слово регистра адреса таблицы
c0tl = c0thi + 4                ;регистр размера таблицы
c0per = c0thi + 6                ;регистр периода звука
c0vol = c0thi + 8               ;регистр громкости
run:                            ;генерация простого тона
    move.l    #table,c0thi      ;записываем начало таблицы
    move      #8,c0tl           ;записываем размер таблицы

```

ВНЕШНИЕ РЕГИСТРЫ

```

move    #400,c0per    ;записываем период тона
move    #40,c0vol     ;записываем громкость
move    #$8201,ctlw   ;запускаем DMA
rts      ;все
data    ;!!!! данные в CHIP-памяти
table:  dc.b    -40,-70,-40,0,40,70,40,0
end
    
```

Загрузите эту программу в отладчик и запустите. Вы услышите ожидаемый звук.

Чтобы остановить звучание, нужно просто обнулить бит 0 регистра DMA. Для этого мы запишем 0 в бит 15 (после чего все установленные в операнде биты будут очищаться в регистре \$DFF096) и 1 в бит 0:

```

still:      ;выключить звук
move    #1,ctlw    ;обнуляем бит 0 регистра ctlw, не
                ;трогая остальные биты
rts      ;и все
    
```

Теперь мы можем написать программу для воспроизведения короткого звука, который, например, можно использовать для индикации нажатия клавиш:

;** Producing a beep tone **

```

ctlw = $DFF096    ;регистр управления DMA
c0thi = $DFF0A0    ;старшее слово регистра адреса таблицы
c0tlo = c0thi + 2  ;младшее слово регистра адреса таблицы
c0tl = c0thi + 4   ;регистр размера таблицы
c0per = c0thi + 6  ;регистр периода звука
c0vol = c0thi + 8  ;регистр громкости
beep:             ;генерация короткого звукового сигнала
move.l    #table,c0thi    ;записываем начало таблицы
move      #8,c0tl         ;записываем размер таблицы
move      #400,c0per      ;записываем период тона
move      #65,c0vol       ;записываем громкость
move      #$8201,ctlw     ;запускаем DMA
move      #20000,D0       ;длительность тона loop:
dbra      D0,loop         ;соответствующая задержка still:
move      #1,ctlw         ;выключаем канал
rts      ;все
table:    dc.b    40,70,90,100,90,70,40,0,-4,0
end
    
```

ВНЕШНИЕ РЕГИСТРЫ

Paula может в обычном режиме воспроизводить до четырех независимых звуковых таблиц (каналы 0-3). Существует и другой интересный режим воспроизведения, в котором некоторые каналы могут использоваться для модуляции других каналов.

В качестве примера рассмотрим программу, генерирующую звук сирены. Конечно, можно это сделать задав полную огибающую звука в одной большой таблице, однако используя режим модуляции такой звук воспроизвести гораздо легче.

Будем использовать два канала: первый — для основного тона и нулевой — для задания огибающей частоты первого канала. На самом деле Paula позволяет задавать не только частотную модуляцию, но и амплитудную (а также обе одновременно), однако на примере мы рассмотрим только частотную модуляцию:

```

** Modulated sound generated via hardware registers **
ctlw = $DFF096          ;регистр управления DMA
adcon = $DFF09E          ;регистр управления звуком
c0thi = $DFF0A0          ;старшее слово регистра адреса таблицы
c0tlo = c0thi + 2        ;младшее слово регистра адреса таблицы
c0tl = c0thi + 4         ;регистр размера таблицы
c0per = c0thi + 6        ;регистр периода звука
c0vol = c0thi + 8        ;регистр громкости
run:                    ;генерация звука сирены
    move.l    #table,c0thi+16 ;адрес таблицы для канала 1
    move      #8,c0tl+16      ;размер таблицы — 8 слов
    move      #300,c0per+16   ;период
    move      #40,c0vol+16    ;громкость
    move.l    #table2,c0thi   ;адрес таблицы для канала 0
    move      #8,c0tl        ;размер таблицы
    move      #60000,c0per    ;период
    move      #30,c0vol       ;громкость
    move      #$8010,adcon     ;включаем режим модуляции по частоте
    move      #$8203,ctlw     ;запускаем DMA
    move.l    #100000,D0      ;счетчик цикла задержки
dl:   subq     #1,D0           ;задержка
      bne.s   dl

```

ВНЕШНИЕ РЕГИСТРЫ

```

still:                                ;** выключить звук
      move    #$10,adcon              ;выключаем режим модуляции
      move    #3,ctlw                 ;выключаем каналы
      rts                                     ;все
table:                                ;данные для основного тона - синусоида
      dc.b    -40,-70,-90,-100,-90,-70,-40,0
      dc.b    40,70,90,100,90,70,40,0
table2:                               ;данные для частотной модуляции, чем
                                     ;выше значение, тем ниже частота
                                     ;основного тона
      dc.w    400,430,470,500,530,500,470,430
      end
  
```

Теперь запустите программу и Вы услышите звук сирены. Изменяя значения в table2 Вы сможете получить другие не менее интересные звуковые эффекты.

В этой программе мы использовали новый регистр — adcon (\$DFF09E). Этот регистр служит для управления звуковыми режимами Paula, а также для работы с диском (на Amiga такие “комбинированные” регистры встречаются довольно часто). В связи с этим запись в adcon осуществляется так же, как и в регистр управления DMA. А именно, происходит наложение битовой маски в режиме, определяемом битом 15, причем нулевые биты маски не влияют на регистр. Как мы уже говорили, это позволяет задавать значения нужных нам битов, не опасаясь за другие биты регистра.

Младшие восемь битов регистра adcon определяют режимы модуляции каналов. Правда существует одно ограничение: любой канал может использоваться для модуляции только следующего (по номеру) канала. Именно поэтому в приведенном примере мы использовали канал 0 для модуляции частоты канала 1. По этой же причине канал 3 не может использоваться для модуляции никакого другого канала.

Следующая таблица показывает назначение битов 0-7 регистра adcon:

ВНЕШНИЕ РЕГИСТРЫ

Бит	Функция
0	Канал 0 модулирует амплитуду канала 1
1	Канал 1 модулирует амплитуду канала 2
2	Канал 2 модулирует амплитуду канала 3
3	Канал 3 выключен
4	Канал 0 модулирует частоту канала 1
5	Канал 1 модулирует частоту канала 2
6	Канал 2 модулирует частоту канала 3
7	Канал 3 выключен

В примере с сиреной мы устанавливаем бит 4, включая тем самым режим частотной модуляции канала 1 каналом 0.

При использовании режимов модуляции некоторые параметры канала приобретают иной смысл. Например, параметр громкости канала, задающего модуляцию, вообще не используется. Также меняется смысл звуковой таблицы модулирующего канала, элементы которой теперь интерпретируются как 16-битные слова. Скорость выборки этих элементов по-прежнему определяется параметром частоты (периода) канала.

При использовании одновременно частотной и амплитудной модуляций элементы таблицы модулирующего канала задаются парами: сначала указывается громкость, а затем частота модулируемого канала.

5.5. Обзор внешних регистров.

В следующих таблицах мы приводим адреса основных регистров hardware Amiga. Ограничения на объем книги не позволяют нам описывать каждый регистр по отдельности; подробную информацию об этих регистрах Вы сможете найти в технической документации по Amiga.

Имейте в виду, что различные эксперименты над внешними регистрами могут привести к сбою или к зависанию компьютера, поэтому не забывайте предварительно сбрасывать всю важную информацию на диск.

Итак, начнем с обзора некоторых регистров CIA (Complex Interface Adaptor):

ВНЕШНИЕ РЕГИСТРЫ

CIA-A	CIA-B	Назначение
BFE001	BFE000	Регистр данных A
BFE101	BFE100	Регистр данных B
BFE201	BFE200	Регистр выбора направления A
BFE301	BFE300	Регистр выбора направления B
BFE401	BFE400	Таймер A (младший байт)
BFE501	BFE500	Таймер A (старший байт)
BFE601	BFE600	Таймер B (младший байт)
BFE701	BFE700	Таймер B (старший байт)
BFE801	BFE800	Регистр событий, биты 0-7
BFE901	BFE900	Регистр событий, биты 8-15
BFEA01	BFEA00	Регистр событий, биты 16-23
BFEB01	BFEB00	Не используется
BFEC01	BFEC00	Регистр последовательного порта
BFED01	BFED00	Регистр управления прерываниями
BFEE01	BFEE00	Регистр управления A
BFEF01	BFEF00	Регистр управления B

Некоторые внутренние функции:

\$BFE101 Регистр данных параллельного порта

\$BFE301 Регистр выбора направления для
параллельного порта

\$BFEC01 Регистр состояния специальных
клавиш клавиатуры

Теперь переходим к обзору звуковых регистров (audio-registers). Будьте осторожны при работе с первыми двумя регистрами, так как они выполняют особенно важные системные функции.

Приводимые регистры доступны либо только для чтения, либо только для записи; мы будем помечать такие регистры как R и W соответственно.

Адрес	Статус	Назначение
DFF096	W	Регистр управления DMA
DFF002	R	Регистр управления DMA (по чтению) и статуса blitter'a

ВНЕШНИЕ РЕГИСТРЫ

— *Аудио-канал 0* —

DFF0AA	W	Регистр (буфер) данных
DFF0A0	W	Регистр начала таблицы данных, биты 16-18
DFF0A2	W	Регистр начала таблицы данных, биты 0-15
DFF0A4	W	Регистр размера таблицы данных
DFF0A6	W	Регистр периода выборки данных
DFF0A8	W	Регистр громкости

— *Аудио-канал 1* —

DFF0BA	W	Регистр (буфер) данных
DFF0B0	W	Регистр начала таблицы данных, биты 16-18
DFF0B2	W	Регистр начала таблицы данных, биты 0-15
DFF0B4	W	Регистр размера таблицы данных
DFF0B6	W	Регистр периода выборки данных
DFF0B8	W	Регистр громкости

— *Аудио-канал 2* —

DFF0CA	W	Регистр (буфер) данных
DFF0C0	W	Регистр начала таблицы данных, биты 16-18
DFF0C2	W	Регистр начала таблицы данных, биты 0-15
DFF0C4	W	Регистр размера таблицы данных
DFF0C6	W	Регистр периода выборки данных
DFF0C8	W	Регистр громкости

— *Аудио-канал 3* —

DFF0DA	W	Регистр (буфер) данных
DFF0D0	W	Регистр начала таблицы

ВНЕШНИЕ РЕГИСТРЫ

		данных, биты 16-18
DFF0D2	W	Регистр начала таблицы
		данных, биты 0-15
DFF0D4	W	Регистр размера таблицы
		данных
DFF0D6	W	Регистр периода выборки
		данных
DFF0D8	W	Регистр громкости

И наконец рассмотрим регистры, содержащие информацию о состоянии джойстика и мыши:

Адрес	R/W	Назначение
DFF00A	R	Порт джойстика/мышь 1
DFF00C	R	Порт джойстика/мышь 2

Раздел 6. ОПЕРАЦИОННАЯ СИСТЕМА.

Настало время перейти к наиболее важной части нашей книги. В предыдущих разделах мы рассмотрели, как писать ассемблерные программы для обработки данных в памяти, однако этого недостаточно для написания “настоящих” программ. До сих пор мы не затрагивали тему организации интерфейса между пользователем и программой, предполагая, что Вы сможете ввести исходные данные и прочитать результаты с помощью отладчика. Из этого раздела Вы узнаете, как выводить текст на экран и вводить данные с клавиатуры (или из файла), а также мы рассмотрим множество других полезных функций, предоставляемых операционной системой Amiga.

Все эти функции реализованы в виде специальных библиотек подпрограмм. Для начала рассмотрим, какие библиотеки входят в состав операционной системы (некоторые из них “защиты” в ROM).

6.1. Загрузка библиотек.

Любая библиотека перед использованием должна быть загружена в память Amiga. К сожалению, даже если Вы планируете использовать только одну функцию, библиотека все равно загружается целиком.

Для начала Вам нужно определить, какие функции должна выполнять программа, с тем, чтобы выбрать необходимые для этого библиотеки. Например, для простого текстового ввода/вывода Вам вовсе не нужно подключать библиотеку для работы с графикой.

На обычном диске Workbench записано несколько стандартных библиотек. Приведем список некоторых из

ОПЕРАЦИОННАЯ СИСТЕМА

них:

exes.library Эта библиотека является самой главной и служит для загрузки других библиотек, а также для выполнения базовых операций, таких, как резервирование памяти и работа с каналами ввода/вывода. Сама библиотека exes находится в ROM.

dos.library Библиотека dos содержит все необходимые функции для организации ввода/вывода: вывод текста на экран либо в файл, итд.

intuition.library Эта библиотека используется для работы с окнами и меню, а также с другими элементами графического интерфейса.

clist.library Содержит функции для работы с соррег-сопроцессором Amiga, который используется для организации экрана.

console.library Содержит подпрограммы ввода/вывода для работы с консолью.

diskfont.library Используется для работы со шрифтами, записанными на диске.

graphics.library Эта библиотека содержит некоторые функции для работы с blitter'ом (графический сопроцессор), а также другие функции для работы с графикой.

icon.library Используется для работы с пиктограммами (icons) Workbench.

layers.library Содержит функции для работы с экранной памятью (layers).

mathfft.library Предоставляет основные математические операции с плавающей точкой.

mathieedoubbas.library Содержит математические функции для работы с целыми числами.

mathtrans.library Содержит некоторые высокоуровневые математические функции.

potgo.library Используется для работы с аналоговыми устройствами ввода.

ОПЕРАЦИОННАЯ СИСТЕМА

timer.library Содержит процедуры для работы с таймерами.
translator.library Содержит единственную функцию "translate", которая служит для перевода текста из обычного представления в фонетическое для использования речевым синтезатором "narrator".

Конечно, Вы можете загрузить все эти библиотеки одновременно, однако необходимо помнить, что на это тратится процессорное время и память Amiga. Поэтому старайтесь как можно более рационально использовать библиотечные ресурсы.

Пусть, например, Вам требуется написать программу, использующую текстовый ввод/вывод. Для этого вам понадобится библиотека "dos.library", которую и нужно открыть (открыть библиотеку — значит в случае необходимости загрузить ее в память). Библиотека "exec.library" содержит функцию OpenLib, которая как раз и используется для открытия других библиотек. Сама библиотека "exec.library" находится в системном ПЗУ (ROM) и открывать ее не надо.

Если открываемая библиотека уже находится в памяти, то она повторно не загружается, обеспечивая таким образом дополнительную экономию памяти и времени. Это означает, что несколько программ могут одновременно использовать одну и ту же библиотеку.

Пакет AssemPro имеет все необходимые файлы для работы с библиотеками, которые содержат, в частности, некоторые полезные макроопределения и символьные константы. Однако, в примерах программ мы не будем использовать эти определения для облегчения переносимости на другие ассемблеры (SEKA, итд.). Правда, в некоторых примерах Вы встретите специфические для AssemPro макросы INIT_AMIGA и EXIT_AMIGA (для обеспечения запуска программ из desktop'a), и если Вы используете другой ассемблер, изучите для начала соответствующую документацию по компоновке программ.

ОПЕРАЦИОННАЯ СИСТЕМА

Если Ваша система “ругается” на макросы AssemPro (ILABEL, INIT_AMIGA, EXIT_AMIGA), то уберите их из программы — обычно это помогает.

6.2. Вызов библиотечных функций.

Для начала опишем механизм вызовов функций библиотек, а затем приведем небольшой пример. Итак, любая библиотека содержит специальную таблицу, которая состоит из команд JMP. Каждая из этих команд осуществляет переход на какую-либо подпрограмму библиотеки. Таким образом, для того, чтобы вызвать n-ную функцию, Вам достаточно вызвать подпрограмму по адресу TABLE + n*SIZE, где TABLE — начало таблицы переходов, а SIZE — размер элементов этой таблицы. Для этого обычно используют команду JSR с нужным смещением. Заметим, что при работе с библиотеками используется адрес конца таблицы переходов, поэтому все смещения имеют отрицательный знак.

Для примера, попробуем открыть библиотеку “dos.library”. Для этого мы должны использовать функцию библиотеки “exec.library”, базовый адрес которой записан в фиксированной ячейке \$000004. Базовый адрес — это адрес таблицы переходов, о которой мы только что говорили.

Для вызова любой функции нам нужно знать ее смещение. Функция OpenLib библиотеки “exec.library” имеет смещение -552 (значения смещений для некоторых функций приведены в приложении в конце книги).

Функция OpenLib требует в качестве параметра адрес ASCII-строки, содержащей имя открываемой библиотеки (в нашем случае “dos.library”). Также нам потребуется ячейка памяти для хранения базового адреса открываемой библиотеки, который возвращается функцией OpenLib. Помните, что имена библиотек должны записываться строчными буквами (dos.library).

Итак, приведем текст программы, открывающей dos.library:

```
,** Load the DOS library 'dos.library' (6.2A) **
```


ОПЕРАЦИОННАЯ СИСТЕМА

```

ExecBase = 4           ;ячейка, содержащая базовый адрес
                        ;библиотеки "exec.library"
OpenLib = -552         ;смещение для функции OpenLib в
                        ;таблице библиотеки exec

init:
    move.l    ExecBase,A6    ;базовый адрес библиотеки exec — в A6
    lea      dosname,A1     ;адрес строки с именем библиотеки dos
    moveq    #0,D0          ;номер версии (0 — любая версия)
    jsr      OpenLib(A6)     ;вызываем OpenLib
    move.l    D0,dosbase     ;сохраняем возвращенный базовый адрес
                        ;библиотеки dos
    beq.s    error          ;если он равен нулю, то произошла
                        ;ошибка
    ...                    ;Ваша программа
    ...                    ;
error:          ;обработка ошибок
    ...
    ...
dosname: dc.b  'dos.library',0 ;имя библиотеки
        align ;even в случае SEKA
dosbase: blk.l 1             ;для хранения адреса открытой
                        ;библиотеки

end
    
```

Таким же образом происходит вызов и других библиотечных функций. Параметры и результаты функций обычно передаются в регистрах, а в случае ошибки в D0 возвращается нулевое значение.

После завершения работы с библиотекой следует выполнить обратную операцию — операцию закрытия этой библиотеки. Для этого используется функция CloseLib, которая имеет смещение -414. Эта функция также содержится в exec.library и требует единственный параметр — базовый адрес закрываемой библиотеки. Например, для закрытия библиотеки "dos.library", нам нужно написать:

```

CloseLib = -414        ;(6.2B)
...
move.l    ExecBase,A6    ;базовый адрес exec — в A6
move.l    dosbase,A1     ;базовый адрес dos — в A1
jsr      CloseLib(A6)    ;закрываем "dos.library"
    
```

ОПЕРАЦИОННАЯ СИСТЕМА

Помните, что открытые библиотеки НЕ закрываются автоматически при завершении Вашей программы. Выход из программ с незакрытыми библиотеками может привести к "замусориванию" памяти Amiga, что, к сожалению, происходит довольно часто. Поэтому не забывайте использовать CloseLib для каждой открытой библиотеки.

6.3. Инициализация программ.

Очевидно, что перед выполнением основной части любой программы нужно выполнить некоторые подготовительные действия. В качестве примера рассмотрим программу редактирования текстов. Для работы подобной программы, во-первых, необходима память для текстового буфера, а во-вторых, окно, воспринимающее ввод с клавиатуры и вывод текста.

Для создания и инициализации всех этих ресурсов используются библиотечные функции. Далее будем предполагать, что библиотека dos.library нами уже открыта.

6.3.1. Резервирование памяти.

Как мы уже говорили, для корректной работы многозадачности пользовательские программы должны резервировать (запрашивать у системы) память перед ее использованием. Это можно сделать несколькими способами.

Рассмотрим наиболее часто используемую для этого функцию. Эта функция находится в библиотеке EXEC и называется AllocMem (смещение - \$C6). AllocMem резервирует область памяти, размер которой указывается в качестве параметра в D0, и возвращает адрес выделенной области в регистре D0 (нулевое значение сигнализирует о том, что система не может найти свободный фрагмент памяти нужного размера).

Функция AllocMem имеет еще и второй параметр (указывается в D1), который определяет режим резервирования (тип выделяемой памяти). Приведем некоторые возможные значения D1:

ОПЕРАЦИОННАЯ СИСТЕМА

D1	Название	Режим резервирования
\$0	MEMF_ANY	Выделить память любого типа
\$2	MEMF_CHIP	Выделить CHIP-память
\$4	MEMF_FAST	Выделить FAST-память
\$10000	MEMF_CLEAR	Очистить память после выделения

Рассмотрим пример:

ExecBase = 4 ;(6.3.1A)

AllocMem = -\$C6

```

...
move.l #number,D0 ;количество байт для резервирования
move.l #mode,D1 ;режим резервирования в D1
move.l ExecBase,A6 ;базовый адрес ЕХЕС
jsr AllocMem(A6) ;захватить память
move.l D0,address ;сохранить адрес выделенной области
beq.s error ;ошибка?
...

```

Другой способ резервирования памяти состоит в использовании функции AllocAbs (смещение -\$CC) вместо AllocMem. Эта функция служит для захвата конкретной области памяти, адрес которой указывается в регистре A1, а размер — в D0. AllocAbs возвращает ноль (в D0), если заданная область памяти не может быть выделена (например, если эта область уже используется какой-либо задачей).

ExecBase = 4 ;(6.3.1B)

AllocAbs = -\$CC

```

...
move.l #number,D0 ;количество байт для резервирования
lea address,A1 ;требуемый адрес блока памяти
move.l ExecBase,A6 ;базовый адрес ЕХЕС
jsr AllocAbs(A6) ;попытка резервирования блока памяти
tst.l D0 ;все ОК?
beq.s error ;ошибка!
...

```

Когда память больше не нужна, ее необходимо “отдать” системе обратно. Это можно сделать с помощью функции FreeMem (смещение -\$D2).

Параметры у этой функции точно такие же, как и у

ОПЕРАЦИОННАЯ СИСТЕМА

AllocAbs (**A1** — адрес освобождаемого блока памяти, **D0** — его размер). Попытка освобождения незарезервированной памяти неизбежно приводит к сбою или зависанию компьютера.

Приведем пример программы, освобождающей память:

```
ExecBase = 4           ;(6.3.1C)
FreeMem = -$D2
```

```
...
move.l    #number,D0    ;размер освобождаемого блока
lea       address,A0     ;адрес освобождаемого блока
move.l    ExecBase,A6    ;базовый адрес EXEC
jsr       FreeMem(A6)    ;освобождаем память
tst.l     D0             ;все ОК?
beq.s     error         ;нет!
...
```

Замечание из предыдущего пункта касается и работы с памятью: операционная система НЕ освобождает память автоматически при завершении программ. Поэтому никогда не забывайте использовать **MemFree** для освобождения всех захваченных блоков памяти.

6.3.2. Создание простого окна ввода/вывода.

В этом пункте мы рассмотрим простейшие средства для создания окон. Вообще, как мы уже упоминали, для работы с окнами и меню используется специальная библиотека — **intuition**, однако ее мы рассмотрим в следующем разделе.

Библиотека **dos** также имеет функцию для создания простых окон, которые служат для обмена текстовой информацией с пользователем. Такие окна называются окнами ввода/вывода операционной системы. Окна ввода/вывода, в отличие от окон **intuition**, не имеют некоторых ресурсов (в частности, отсутствует символ закрытия (**close gadget**) в левом верхнем углу окна).

Итак, для создания окна ввода/вывода необходимо использовать функцию **Open** библиотеки **dos.library**. Вообще, эта функция используется настолько часто, что ее вы-

ОПЕРАЦИОННАЯ СИСТЕМА

зов целесообразно поместить в отдельную подпрограмму:

;** Load the DOS library 'dos.library' (6.3.2A) **

```

ExecBase      = 4           ;ячейка, в которой содержится
                           ;базовый адрес библиотеки ehex
OpenLib        = -552       ;смещение функции
OpenLib Open   = -30       ;смещение функции Open
init:
    move.l     ExecBase,A6   ;базовый адрес ehex — в A6
    lea        dosname(PC),A1 ;адрес строки названия библиотеки
    moveq      #0,D0         ;номер версии: не имеет значения
    jsr        OpenLib(A6)   ;открываем dos.library
    move.l     D0,dosbase    ;сохраняем ее базовый адрес
    beq.s      error        ;ошибка открытия
    ...                ;Ваша программа
    ...                ;...продолжается
    ...                ;...здесь
error:
    ...                ;Ваш обработчик ошибок
OpenFile:
    ...                ;Функция Open, которая
    ...                ;используется не
    ...                ;только для создания окон
    ...                ;ввода/вывода, но и для создания
    ...                ;файлов
    move.l     dosbase,A6    ;извлекаем адрес библиотеки dos
    jsr        Open(A6)      ;вызываем Open
    tst.l      D0            ;ОК? проверка — после возврата
    rts        ;возврат
dosname:
    dc.b       'dos.library',0
    align      2             ;выравниваем адрес по четности
dosbase:
    ...                ;место для хранения
    ...                ;адреса dos.library
    blk.l      1

```

Но это еще не все. Функция Open требует несколько параметров, уточняющих ее действия. Эти параметры передаются на регистрах D1 и D2. D1 указывает на строку, содержащую имя открываемого файла (эта строка должна оканчиваться нулевым байтом), а D2 содержит 32-битное слово, задающее режим работы с файлом.

ОПЕРАЦИОННАЯ СИСТЕМА

Теперь поясним, как все это можно применить для открытия окна ввода/вывода. К счастью, операционная система Amiga позволяет одинаковым образом работать с файлами, консолью (клавиатурой и экраном) и с портом RS232, используя обобщенное понятие канала ввода/вывода.

Нас интересует именно консоль, так как при указании ее в качестве имени файла система сама откроет окно ввода/вывода. Имя консоли начинается с 'CON:' подобно тому, как имена файлов на диске имеют префикс 'DF0:'. В дополнение к этому нужно указать координаты X и Y левого верхнего угла окна, а также его размеры и заголовок. Полное определение имени консоли для создания окна выглядит примерно так:

```
consolname: dc.b 'CON:0/100/640/100/*-Window-*-',0
```

Эта строка определяет окно, левый верхний угол которого имеет координаты X = 0 и Y = 100, а ширина и высота равны соответственно 640 и 100 точек.

Чтобы открыть такое окно, нужно написать:

```
mode_old = 1005                                ;режим открытия файла консоли
        lea     consolname(PC),A1
        move.l  A1,D1                          ;параметр нужен в D1
        move.l  #mode_old,D2
        bsr     openfile                      ;вызов нашей
                                                ;подпрограммы (см. выше)
        beq.s   error                        ;обработка ошибок
        move.l  D0,conhandle
        rts
        ...
conhandle: dc.l  1                            ;место для идентификатора
                                                ;консоли
```

Сделаем несколько замечаний:

Режим mode_old обычно используется для открытия файлов, которые уже существуют, однако он же используется и для работы с консолью, хотя это и приводит к созданию нового окна.

Если при открытии файла произошла ошибка (например, файл не найден), то в регистре D0 возвращается

ОПЕРАЦИОННАЯ СИСТЕМА

ноль, иначе возвращается идентификационный номер созданного канала ввода/вывода. Этот номер необходимо сохранить, так как он должен передаваться в качестве параметра всем функциям, работающим с каналами ввода/вывода. В нашем примере этот номер сохраняется в переменной `conhandle`.

Как уже упоминалось, окна ввода/вывода не имеют символа закрытия (`close gadget`), однако функции изменения размера и размещения на переднем/заднем плане существуют и для них. Заботу об этих функциях полностью берет на себя операционная система (в отличие от компьютеров ATARI ST, где изменение размера должно контролироваться программой пользователя).

Как и в случае с библиотеками, открытые каналы ввода/вывода нужно закрывать до завершения программы. Это делается с помощью функции `Close`, которая имеет смещение -36. Функция `Close` вызывается с единственным параметром в регистре `D1`, который должен содержать идентификатор закрываемого канала.

Таким образом, в конце программы нам нужно поместить следующий фрагмент:

```
Close = -36 ;(6.3.2C)
```

...

```
move.l conhandle,D1 ;идентификатор канала консоли
move.l dosbase,A6 ;базовый адрес библиотеки dos
jsr Close(A6) ;закрываем канал
```

После этого окно исчезнет.

Вы можете создавать несколько окон, выполняя по одному вызову `Open` на каждое окно. В этом случае Вы будете иметь несколько идентификаторов, с каждым из которых можно работать по отдельности.

Приведем законченную программу, работающую с окном ввода/вывода. В этой программе мы используем макросы `INIT_AMIGA` и `EXIT_AMIGA`, предоставляемые системой `AssemPro` для выполнения входной и выходной инициализации; если Вы используете другую систему (например,

ОПЕРАЦИОННАЯ СИСТЕМА

SEKA), изучите соответствующую документацию.

;**** 6.3.2 S.D. ****

;* Функции Exec:

OpenLib = -552

CloseLib = -414

;ExecBase = 4 ;описано в AssemPro

;* Функции AmigaDos:

Open = -30

Close = -36

IoErr = -132 ;получение дополнительной
;информации об ошибке

mode_old = 1005

ILABEL Assempro:includes/Amiga.l ;только для AssemPro!

INIT_AMIGA ;только для AssemPro

run:

```

    bsr.s    init          ;инициализация
                                ;(создание окна)
    bra.s    qu            ;выход init:
    move.l   ExecBase,A6   ;базовый адрес EXEC
    lea      dosname(PC),A1 ;'dos.library'
    moveq    #0,D0         ;годится любая версия
    jsr      OpenLib(A6)   ;открываем dos.library
    move.l   D0,dosbase
    beq.s    error1        ;ошибка
    lea      consolname(PC),A1 ;описание окна
    move.l   #mode_old,D0
    bsr.s    openfile      ;открываем консоль
    beq.s    error         ;ошибка
    move.l   D0,conhandle
    rts

```

```

error:
    move.l   dosbase,A6    ;обработчик ошибок DOS
    jsr      IoErr(A6)

```

```

error1:
    move.l   D0,D5         ;нужно для EXIT_AMIGA
    move.l   #-1,D7        ;нужно для EXIT_AMIGA

```

```

qu:
    move.l   conhandle(PC),D1
    move.l   dosbase(PC),A6
    jsr      Close(A6)     ;закрываем консоль

```


ОПЕРАЦИОННАЯ СИСТЕМА

```

move.l   dosbase(PC),A1
move.l   ExecBase,A6
jsr      CloseLib(A6)           ;закрываем dos.library
EXIT_AMIGA                       ;только для AssemPio!
openfile:
move.l   A1,D1                 ;открыть канал ввода/вывода
move.l   D0,D2                 ;имя файла — в D1
move.l   dosbase(PC),A6       ;режим — в D2
jsr      Open(A6)              ;открываем файл (консоль)
tst.l    D0                    ;ошибка?
rts

dosname:  dc.b    'dos.library',0
align.w
dosbase:  dc.l    0
consoName: dc.b    'CON:0/100/640/100/** CLI-Test **',0
align.w
conhandle: dc.l    0
end
    
```

Есть еще один способ открыть окно ввода/вывода, а именно, вместо CON: можно использовать RAW:. Все остальные параметры в имени консоли останутся прежними. Если Вы в предыдущей программе исправите CON: на RAW:, то внешний вид окна не изменится. Разница проявляется только при вводе данных с клавиатуры: например, в случае CON: клавиши управления курсором работают как обычно, а в случае RAW: игнорируются.

6.4. Ввод/вывод.

Помимо основных действий по обработке данных, любая программа должна осуществлять обмен информацией (например, с пользователем). Для этого используются специальные функции операционной системы — функции ввода/вывода. Эти функции позволяют работать с принтером, клавиатурой, параллельным и последовательным портами, а также с дисководом и другими устройствами.

В этом пункте мы рассмотрим основные методы организации ввода/вывода и приведем все необходимые подпрограммы. Объединив эти подпрограммы в одну библиоте-

ОПЕРАЦИОННАЯ СИСТЕМА

ку, Вы сможете в последствии использовать их при написании собственных программ. В конце каждого подпункта мы будем приводить примеры законченных программ, использующих функции ввода/вывода.

Перед тем, как осуществлять ввод/вывод, нужно приготовить данные для вывода в требуемом формате, а также зарезервировать память для ввода данных. К этому моменту библиотека dos.library должна быть уже открыта.

Многие программы при запуске выводят некоторый текст на экран (например, заголовок или приглашение ко вводу данных с клавиатуры). Именно с этого мы и начнем рассмотрение функций библиотеки dos.library.

6.4.1. Вывод текста на экран.

Для компьютеров, подобных Amiga, сразу возникает вопрос: а куда следует выводить текст? Компьютеры, работающие только с одним экраном, избавлены от этой "проблемы". Нам же понадобится сначала определить окно, в которое текст должен выводиться.

У нас есть две основные возможности:

1. Выводить текст в окно интерпретатора командной строки (CLI)

2. Выводить текст в любое другое окно

В первом случае вывод будет работать только в случае, если программа запущена из CLI. Иначе Вам придется открыть собственное окно.

Чтобы обеспечить работу программы в любых условиях, будем открывать окно ввода/вывода (консоль). Существует несколько режимов работы с консолью (например, RAW: или CON:), но мы для определенности будем использовать только CON:.

Итак, допустим, что Вы открыли окно ввода/вывода и хотите напечатать в нем текстовую строку. Для использования в функции вывода эта строка должна быть представлена в следующем виде:

```
title:      dc.b      "*** Welcome to this program ! ***"
titleend:
```

ОПЕРАЦИОННАЯ СИСТЕМА

align ;или even

Мы рекомендуем всегда использовать директиву align (even) после определения текстовой строки, так как это гарантирует корректность адресов при дальнейших описаниях.

Для вывода текста в созданное окно используется функция Write. Эта функция имеет смещение -48 и вызывается с тремя параметрами:

в D1 идентификатор канала (консоли) для вывода;

в D2 адрес строки текста для вывода (в нашем случае адрес "title");

в D3 количество байт для вывода.

Чтобы определить число байт для вывода, нужно подсчитать количество символов в тексте. Однако за Вас это может сделать ассемблер, если Вы напишете:

```
move.l #titleend-title,D3
```

Главным преимуществом такого определения является то, что Вы легко сможете изменять текст, не заботясь о его длине, а также добавлять некоторые управляющие коды (спецсимволы).

Рассмотрим пример использования Write:

```
Write    = -48                ;(6.4.1A)
...      ;здесь нужно открыть окно
...
move.l   dosbase,A6          ;адрес библиотеки dos
move.l   conhandle,D1        ;идентификатор консоли
move.l   #title,D2           ;указатель на строку для вывода
move.l   #titleend-title,D3   ;количество символов
jsr      Write(A6)           ;вызываем функцию
...
title:   dc,b                "*** Welcome to this program ! ***"
titleend:
align    ;или even (для SEKA)
end
```

Это, несомненно, очень полезная функция. Однако, часто бывает нужно, например, вывести всего один символ, за-

ОПЕРАЦИОННАЯ СИСТЕМА

писанный в регистре `dachnyx`. Поскольку такой функции вывода не существует, нам придется написать специальную подпрограмму, реализующую посимвольный вывод через функцию `Write`. В общем случае Вам могут понадобиться следующие подпрограммы:

```

pmsg  выводит текст, начиная с (D2), до первого
      нулевого байта;

pline  то же самое, только в конце строки
      добавляются специальные символы для
      перевода курсора на следующую строку;

pchar  выводит единственный символ, код которого
      записан в D0;

pcrlf  переводит курсор на следующую строку.
Приведем тексты этих подпрограмм:

Write  = -48                      ;(6.4.1B)
      ...

pline:  ;* вывод строки с переводом курсора
      bsr.s    pmsg

pcrlf:  ;* перевод курсора на следующую строку
      move     #10,D0             ;код LF (перевод строки)
      bsr.s    pchar             ;выводим
      move     #13,D0             ;код CR (возврат каретки)

pchar:  ;* вывод символа с кодом в D0
      move.b   D0,outline         ;символ — в буфер вывода
      move.l   #outline,D2        ;адрес буфера — в D2

pmsg:   ;* вывод строки до нулевого байта
      move.l   D2,A0              ;адрес — в A0
      moveq    #0,D3              ;начальная длина — в D3

ploop:  ;
      tst.b    (A0)+              ;нулевой байт?
      beq.s    pmsg2             ;да, конец
      addq.l   #1,D3              ;иначе увеличиваем длину строки
      bra.s    ploop             ;цикл

pmsg2:  ;
      move.l   dosbase,A6         ;базовый адрес dos.library — в A6
      move.l   conhandle,D1       ;идентификатор канала — в D1
      jsr      Write(A6)          ;вызываем Write
      rts                      ;и все!
    
```

ОПЕРАЦИОННАЯ СИСТЕМА

```
outline:    dc.w    0           ;буфер для единичного символа
conhandle:  dc.l    0           ;идентификатор окна (консоли)

    Теперь приведем пример программы, которая открывает простое окно и печатает в нем текстовую строку (здесь мы также используем макросы системы AssemPro INIT_AMIGA и EXIT_AMIGA):
:***** 6.4.1C.asm S.D. ***** (очередная "кривая" программа)
OpenLib    = -552
CloseLib   = -414
;ExecBase  = 4                  ;в AssemPro уже определено
;* Функции AmigaDos:
Open       = -30
Close      = -36
IoErr      = -132              ;получение дополнительной
                                ;информации об ошибках DOS

Write      = -48
mode_old   = 1005
ILABEL     Assempro:includes/Amiga.l ;только для AssemPro!
INIT_AMIGA ;только для AssemPro
run:
    bsr.s    init              ;создание окна итд.
    bsr      test              ;основная программа
    nop                      ;одному автору известно, зачем
                                ;здесь нужен nop
    bra.s    qu                ;выход из программы

test:
    move.l    #title,D0
    bsr.s     pmsg              ;печать текста в окне
    bsr.s     pcrlf             ;перевод строки
    rts       ;возврат init:
    move.l    ExecBase,A6       ;базовый адрес EXEC
    lea       dosname(PC),A1    ;'dos.library'
    moveq     #0,D0             ;годится любая версия
    jsr       OpenLib(A6)       ;открываем dos.library
    move.l    D0,dosbase
    beq.s     error1            ;ошибка
    lea       consolname(PC),A1 ;описание окна
    move.l    #mode_old,D0
    bsr.s     openfile          ;открываем консоль
```

ОПЕРАЦИОННАЯ СИСТЕМА

	beq.s	error	;ошибка
	move.l	D0,conhandle	
	rts		
pmsg:			;печатать сообщения по (D0)
	movem.l	d0-d7/a0-a6,-(SP)	
	move.l	D0,A0	
	move.l	A0,D2	
	moveq	#0,D3	
ploop:			;цикл вычисления длины строки
	tst.b	(A0)+	
	beq.s	pmsg2	
	addq.l	#1,D3	
	bra.s	ploop	
pmsg2:			
	move.l	conhandle(PC),D1	
	move.l	dosbase(PC),A6	
	jsr	Write(A6)	;вывод в окно
	movem.l	(SP)+,d0-d7/a0-a6	
	rts		
pCrLf:			;вывод спецсимволов LF и CR
	move	#10,D0	
	bsr	pchar	
	move	#13,D0	
pchar:			;вывод символа по коду в D0
	movem.l	d0-d7/a0-a6,-(SP)	
	move.l	conhandle(PC),D1	
pchl:			
	lea	outline(PC),A1	
	move.b	d0,(A1)	
	move.l	A1,D2	
	move.l	#1,D3	;выводим только 1 символ
	move.l	dosbase(PC),A6	
	jsr	Write(A6)	
	movem.l	(SP)+,d0-d7/a0-a6	
rts			
error:	move.l	dosbase,A6	;обработчик ошибок DOS
	jsr	IoErr(A6)	
error1:			
	move.l	D0,D5	;нужно для EXIT_AMIGA
	move.l	#-1,D7	;нужно для EXIT_AMIGA

ОПЕРАЦИОННАЯ СИСТЕМА

```

qu:
    move.l    conhandle(PC),D1
    move.l    dosbase(PC),A6
    jsr       Close(A6)                ;закрываем консоль
    move.l    dosbase(PC),A1
    move.l    ExecBase,A6
    jsr       CloseLib(A6)            ;закрываем dos.library
    EXIT_AMIGA                         ;только для AssemPro!

openfile:
    ;открыть канал ввода/вывода
    move.l    A1,D1                  ;имя файла — в D1
    move.l    D0,D2                  ;режим — в D2
    move.l    dosbase(PC),A6
    jsr       Open(A6)               ;открываем файл (консоль)
    tst.l     D0                     ;ошибка?
    rts

dosname:     dc.b    'dos.library',0
             align.w

dosbase:     dc.l    0

consolname:  dc.b    'CON:0/100/640/100/** CLI-Test **',0
             align.w

conhandle:   dc.l    0

title:       dc.b    '*** Welcome to this program! **',0
             align.w

outline:     dc.w    0
             end
    
```

Нетрудно заметить, что приведенные в этом пункте подпрограммы (prmsg, pline, итд.) отличаются особой “кривизной” (почему бы не использовать один буфер сразу для двух символов — LF и CR?). Предоставим написание более “правильной” версии читателю в качестве упражнения...

Как мы уже упоминали, с помощью Write можно выводить специальные символы (special characters). Некоторые из них мы уже использовали в примерах (LF и CR). Такие символы называются управляющими, так как они никак не отображаются на экране, но исполняют некоторые действия по управлению выводом. Например, вывод символа LF вызывает перевод курсора на одну строку вниз (при этом номер столбца (X-координата) курсора не меняется).

ОПЕРАЦИОННАЯ СИСТЕМА

Приведем список некоторых управляющих символов и последовательностей (коды символов записаны в шестнадцатеричной системе):

Код	Функция
08	"забой" (backspace) — удаление предыдущего символа
0A	перевод курсора на одну строку вниз (LF)
0B	перевод курсора на одну строку вверх
0C	очистка экрана
0D	перевод курсора в начало текущей строки (CR)
0E	Выключить режим спецсимволов (см. 0F)
0F	Включить режим спецсимволов 1B escape

Существуют, также, управляющие последовательности, которые начинаются с кода CSI (Control Sequence Introductor, код \$9B). Символы, следующие за CSI, определяют конкретную функцию. В следующей таблице [n] обозначает десятичное число, записанное в ASCII-виде. Это число может быть опущено, в этом случае будет использоваться значение в круглых скобках.

Последовательность	Функция
9B [n] 40	вставить n пробелов
9B [n] 41	передвинуть курсор на n (1) строк вверх
9B [n] 42	передвинуть курсор на n (1) строк вниз
9B [n] 43	передвинуть курсор на n (1) символов вправо
9B [n] 44	передвинуть курсор на n (1) символов влево
9B [n] 45	установить курсор в первый столбец и передвинуть на n (1) строк вниз
9B [n] 46	установить курсор в первый столбец и передвинуть на n (1) строк вверх
9B [n] [3B m] 48	установить курсор на строку n и в

ОПЕРАЦИОННАЯ СИСТЕМА

	столбец m
9B 4A	очистить экран от курсора до конца
9B 4B	очистить строку от курсора до конца
9B 4C	вставить строку
9B 4D	удалить строку
9B [n] 50	удалить n символов, начиная от курсора
9B [n] 53	прокрутить экран вверх на n строк
9B [n] 54	прокрутить экран вниз на n строк
9B 32 30 68	интерпретировать перевод строки, как LF + CR
9B 32 30 6C	интерпретировать перевод строки, как LF
9B 6E	получить координаты курсора в виде: 9B (строка) 3B (столбец) 52
9B (style);(fcolor);(bcolor) 6D	задать режим вывода. Параметр style может принимать значения: 0 = нормальный режим 1 = жирный шрифт 3 = наклонный шрифт 4 = режим подчеркивания 7 = режим инверсии Параметр fcolor задает цвет символов и может принимать значения от 30 до 37 (цвета 0-7); параметр bcolor задает цвет фона и может принимать значения от 40 до 47 (цвета 0-7). Все параметры задаются в ASCII-формате в десятичной системе счисления.
9B (Length) 74	установить максимальное количество отображаемых строк
9B (Width) 75	установить максимальную длину строки

ОПЕРАЦИОННАЯ СИСТЕМА

9B (Distance) 78	установить расстояние (в пикселах) от левой границы окна до выводимого текста
9B (Distance) 79	установить расстояние (в пикселах) от верхней границы окна до выводимого текста Если в последних четырех функциях опустить параметр, то будут установлены значения по умолчанию.
9B 30 20 70	убрать курсор
9B 20 70	показать курсор
9B 71	получить конфигурацию окна в виде: 9B 31 3B 31 3B (строка) 3B (столбцов) 73

Текст с использованием спецсимволов может выглядеть, например, так:

```
mytext: dc.b $9B,"4;31;40",$6D      ;включаем режим подчеркивания
        dc.b "Underline"
        dc.b $9B,"3;33;40",$6D      ;включаем наклонный шрифт
        dc.b $9B,"5;20",$48         ;устанавливаем курсор в
                                     ;позицию 5,20
        dc.b "*** Hello, world! ***",0
```

Так как параметры для управляющих последовательностей представляют собой ASCII-строки, мы записываем их в кавычках.

А теперь приведем полный текст программы, которая создает окно и печатает в нем ASCII-строку, содержащую управляющие коды:

```
;***** 6.4.1D.asm S.D. *****
OpenLib      = -552
CloseLib     = -414
;ExecBase    = 4

Open         = -30
Close        = -36
IoErr        = -132
Write        = -48
```

ОПЕРАЦИОННАЯ СИСТЕМА

mode_old = 1005

```

ILABEL      Assempro:includes/Amiga.l      ;только для AssemPro
INIT_AMIGA  ;только для AssemPro
run:
    bsr.s    init                          ;создание окна итд.
    bsr.s    test                          ;основная программа
    nop                                             ;одиному автору известно,
                                             ;зачем здесь нужен nop
    bra.s    qu                            ;выход

test:
    move.l   #mytext,D0
    bsr.s    pmsg                          ;печать текста в окне
    bsr.s    pcrlf                         ;перевод строки
    bsr.s    pcrlf                         ;еще раз перевод строки
    rts                                             ;возврат

init:
    move.l   ExecBase,A6                  ;базовый адрес EXEC
    lea      dosname(PC),A1               ;'dos.library'
    moveq    #0,D0                        ;годится любая версия
    jsr      OpenLib(A6)                  ;открываем dos.library
    move.l   D0,dosbase
    beq.s    error1                       ;ошибка
    lea      consolname(PC),A1            ;описание окна
    move.l   #mode_old,D0
    bsr.s    openfile                     ;открываем консоль
    beq.s    error                         ;ошибка
    move.l   D0,conhandle
    rts

pmsg:
                                             ;печать сообщения по (D0)
    movem.l  d0-d7/a0-a6,-(SP)
    move.l   D0,A0
    move.l   A0,D2
    moveq    #0,D3

ploop:
    tst.b    (A0)+
    beq.s    pmsg2
    addq.l   #1,D3
    bra.s    ploop

pmsg2:

```

ОПЕРАЦИОННАЯ СИСТЕМА

```

move.l  conhandle(PC),D1
move.l  dosbase(PC),A6
jsr      Write(A6)                ;вывод в окно
movem.l  (SP)+,d0-d7/a0-a6
rts

pcrlf:
move     #10,D0                  ;вывод спецсимволов LF и CR
bsr      pchar
move     #13,D0

pchar:
movem.l  d0-d7/a0-a6,-(SP)
move.l   conhandle(PC),D1

pch1:
lea      outline(PC),A1
move.b   d0,(A1)
move.l   A1,D2
move.l   #1,D3                   ;выводим только 1 символ
move.l   dosbase(PC),A6
jsr      Write(A6)
movem.l  (SP)+,d0-d7/a0-a6
rts

error:
move.l   dosbase,A6              ;обработчик ошибок DOS
jsr      IoErr(A6)

error1:
move.l   D0,D5                   ;для EXIT_AMIGA
move.l   #-1,D7                  ;для EXIT_AMIGA (флаг)

qu:
move.l   conhandle(PC),D1
move.l   dosbase(PC),A6
jsr      Close(A6)               ;закрываем консоль
move.l   dosbase(PC),A1
move.l   ExecBase,A6
jsr      CloseLib(A6)            ;закрываем dos.library
EXIT_AMIGA                       ;только для AssemPro!

openfile:
move.l   A1,D1                   ;открыть канал ввода/вывода
move.l   D0,D2                   ;имя файла — в D1
move.l   dosbase(PC),A6          ;режим — в D2
jsr      Open(A6)                ;открываем файл (консоль)
tst.l    D0                      ;ошибка?

```

ОПЕРАЦИОННАЯ СИСТЕМА

```

rts
dosname:   dc.b      'dos.library',0
           align.w
dosbase:   dc.l      0
console:   dc.b      'CON:0/100/640/100/** CLI-Test **',0
           align.w
conhandle: dc.l      0
mytext:    dc.b      $9B,"4;31;40",$6D
           dc.b      "Underline"
           dc.b      $9B,"3;33;40",$6D
           dc.b      $9B,"5;20",$48
           dc.b      "
outline:    dc.w      0
           end

```

;используем режим подпрограммы
 ;включаем наклонный шрифт
 ;устанавливаем курсор в
 ;позицию 5,20

Итак, мы рассмотрели все основные функции вывода текста. Теперь переходим к функциям ввода.

6.4.2. Ввод с клавиатуры.

Читать данные с клавиатуры совсем несложно. Для этого нужно просто открыть консоль (окно ввода/вывода) и читать данные из полученного канала. В предыдущем пункте мы рассмотрели функцию Write, которая используется для записи данных в канал. Библиотека DOS содержит подобную функцию и для чтения из канала. Эта функция называется Read и имеет смещение -42.

Функция Read, как и Write, вызывается с тремя параметрами:

- в D1 идентификатор канала ввода/вывода
- в D2 адрес буфера для чтения данных
- в D3 число байт для чтения

Рассмотрим подпрограмму, которая вводит несколько символов с клавиатуры в некоторый буфер:

```

Read      = -42                      ;(6.4.2A)
...
getchr:   ;получить (D3) символов с клавиатуры
           move.l   #inbuff,D2       ;буфер ввода — в D2
           move.l   dosbase(PC),A6   ;базовый адрес библиотеки dos.library
           jsr      Read(A6)         ;вызов функции чтения из канала

```

ОПЕРАЦИОННАЯ СИСТЕМА

```

        rts                                ;и все!
inbuff: blk.b 80,0                        ;буфер ввода с клавиатуры

```

Возврат из функции Read происходит при нажатии <Enter>. Если при этом количество введенных символов больше, чем требовалось (в D3), то в буфер записываются только первые (D3) символов, а остальные символы могут быть прочитаны при следующем обращении к Read.

Вводимые символы автоматически печатаются на экране, причем для редактирования строки Вы можете использовать клавишу "Backspace". Число символов, записанных в буфер inbuff, возвращается функцией Read в D0.

Сказанное относится только к консоли типа CON:. RAW:- консоль работает несколько по-другому, см. далее.

Рассмотрим программу, в которой за созданием окна ввода/вывода следует фрагмент:

```

        move #80,D3
        bsr.s getchr                      ;читаем 80 символов
        lea  inbuff,A0
        clr.b 0(A0,D0)                   ;добавляем нулевой байт в конце
        bsr  pmsg                         ;выводим содержимое буфера
bp:      nop                             ;место для контрольной точки

```

Загрузив эту программу в отладчик и поставив контрольную точку на адрес "bp" (в системе SEKA для этого нужно ввести "bp" после запуска программы), Вы сможете наблюдать за работой функции Read:

После запуска программы на экране появится окно с курсором в левом верхнем углу. Введите с клавиатуры какой-нибудь текст и нажмите <Enter> — Вы увидите, что введенная Вами строка напечатается второй раз.

Мы использовали подпрограмму "pmsg" из предыдущего пункта, которая выводит ASCII-строку, оканчивающуюся нулевым байтом. Для добавления нулевого байта в конце строки мы используем команду clr.b: так как D0 содержит число прочитанных байт, то 0(A0,D0) как раз указывает на первую неиспользованную ячейку буфера.

Распечатав в отладчике содержимое буфера inbuff после ввода текста, Вы увидите, что введенная Вами стро-

ОПЕРАЦИОННАЯ СИСТЕМА

ка заканчивается кодом \$A. Это — код клавиши <Enter>, который тоже учитывается при вводе. Поэтому, например, после ввода строки "12" Вы обнаружите, что D0 содержит число 3.

А теперь исправьте в имени консоли "CON:" на "RAW:" и запустите программу. Вы сразу же заметите разницу. Теперь функция Read будет возвращать управление после ввода каждого символа, не дожидаясь <Enter>. При этом D0 будет содержать 1.

Преимуществом такого режима ввода является возможность работы с клавишами управления курсором. Написав специальную подпрограмму, Вы можете вводить данные посимвольно с помощью getch и обрабатывать управляющие коды.

Библиотека dos.library предоставляет еще одну полезную функцию для работы с клавиатурой — WaitForChar (смещение -204). Эта функция ждет нажатия любой клавиши в течение заданного интервала времени, и если клавиша была нажата, то WaitForChar возвращает -1 (\$FFFFFFFF), иначе — 0. Эта функция может использоваться, например, для приостановки вывода текста и ожидания нажатия любой клавиши для продолжения.

Параметры функции WaitForChar:

в D1 идентификатор канала

в D2 длительность ожидания нажатия клавиши (в микросекундах)

Например, следующая программа ждет нажатия клавиши в течение 1 секунды:

WaitForCh = -204 ; (6.4.2C)

```
...
scankey:                                ; * ждет нажатия клавиши
      move.l    conhandle(PC),D1        ; идентификатор окна (консоли)
      move.l    #1000000,D2             ; 1 сек. = 1000000 микросекунд
      move.l    dosbase(PC),A6          ; база библиотеки dos.library
      jsr       WaitForCh(A6)           ; ждем клавишу
      tst.l     D0                      ; проверка нажатия
      rts                                ; возврат
```

ОПЕРАЦИОННАЯ СИСТЕМА

Здесь команда `tst` используется для проверки результата работы `WaitForChar`, так что сразу после вызова "scankey" основная программа может выполнить ветвление (`BNE` или `BEQ`).

Приведем пример программы, которая открывает окно, печатает в нем некоторый текст и ждет ввода с клавиатуры.

```
;***** (6.4.2A).ASM S.D. *****
```

```
OpenLib      = -552
CloseLib     = -414
;ExecBase    = 4
```

```
Open         = -30
Close        = -36
IoErr        = -132
Read         = -42
Write        = -48
mode_old     = 1005
```

```
ILABEL      Assempro:includes/Amiga.l ;только для AssemPro
INIT_AMIGA  ;только для AssemPro
run:
    bsr.s    init                    ;создание окна итд.
    bsr.s    test                    ;основная программа
    nop      ;???
    bra.s    qu                      ;выход

test:
    move.l   #mytext,D0
    bsr      pmsg                    ;печать текста в окне
    bsr      pcrlf                   ;перевод строки
    bsr      pcrlf                   ;еще раз перевод строки
    moveq    #80,D3                  ;80 символов для чтения
    bsr      getchr                  ;ввод строки
    bsr      pchar                    ;печать результата
    rts      ;возврат

init:
    move.l   ExecBase,A6              ;базовый адрес EXEC
    lea      dosname(PC),A1          ;'dos.library'
    moveq    #0,D0                    ;годится любая версия
```


ОПЕРАЦИОННАЯ СИСТЕМА

```

jsr      OpenLib(A6)           ;открываем dos.library
move.l   D0,dosbase
beq.s    error1                ;ошибка
lea      consolname(PC),A1     ;описание окна
move.l   #mode_old,D0
bsr.s    openfile              ;открываем консоль
beq.s    error                  ;ошибка
move.l   D0,conhandle
rts

pmsg:
movem.l  d0-d7/a0-a6,-(SP)
move.l   D0,A0
move.l   A0,D2
moveq    #0,D3

ploop:
tst.b    (A0)+
beq.s    pmsg2
addq.l   #1,D3
bra.s    ploop

pmsg2:
move.l   conhandle(PC),D1
move.l   dosbase(PC),A6
jsr      Write(A6)             ;вывод в окно
movem.l  (SP)+,d0-d7/a0-a6
rts

pcrlf:
move     #10,D0                ;вывод спецсимволов LF и CR
bsr      pchar
move     #13,D0

pchar:
movem.l  d0-d7/a0-a6,-(SP)
move.l   conhandle(PC),D1
;вывод символа по коду в D0

pch1:
lea      outline(PC),A1
move.b   d0,(A1)
move.l   A1,D2
move.l   #1,D3                ;выводим только 1 символ
move.l   dosbase(PC),A6
jsr      Write(A6)
movem.l  (SP)+,d0-d7/a0-a6
    
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

    rts
getchr:                                ;ввод символа с клавиатуры
    move.l    conhandle(PC),D1
    lea       inbuff(PC),A1           ;буфер ввода
    move.l    A1,D2
    move.l    dosbase(PC),A6
    jsr       Read(A6)
    move.l    #inbuff,D0              ;возвращаем адрес строки
    rts

error:
    move.l    dosbase,A6               ;обработчик ошибок DOS
    jsr       IoErr(A6)

error1:
    move.l    D0,D5                    ;для EXIT_AMIGA
    move.l    #-1,D7                   ;для EXIT_AMIGA (флаг)

qu:
    move.l    conhandle(PC),D1
    move.l    dosbase(PC),A6
    jsr       Close(A6)                ;закрываем консоль
    move.l    dosbase(PC),A1
    move.l    ExecBase,A6
    jsr       CloseLib(A6)             ;закрываем dos.library
    EXIT_AMIGA                         ;только для AssemPro!

openfile:
    ;открыть канал ввода/вывода
    move.l    A1,D1                    ;имя файла — в D1
    move.l    D0,D2                    ;режим — в D2
    move.l    dosbase(PC),A6
    jsr       Open(A6)                 ;открываем файл (консоль)
    tst.l     D0                       ;ошибка?
    rts

dosname:
    dc.b      'dos.library',0
    align.w
dosbase:     dc.l      0
consolname:  dc.b      'CON:0/100/640/100/** CLI-Test **',0
    align.w
conhandle:   dc.l      0
mytext:      dc.b      "*** Hello, world! **",0
    align.w
outline:     dc.w      0
    
```

ОПЕРАЦИОННАЯ СИСТЕМА

```
inbuff:    blk.b    80  
          end
```

6.4.3. Работа с принтером.

До сих пор мы рассматривали ввод/вывод на примере консоли. Теперь поговорим о других устройствах вывода. Для начала рассмотрим, как Amiga работает с принтером.

Для принтера выделяется специальный тип каналов ввода/вывода — “PRT:”, подобно тому, как для консоли выделяются каналы “CON:” и “RAW:”. Правда, в отличие от консоли, канал принтера должен открываться в режиме `mode_new = 1006`. А дальше все очень просто: Вы используете уже известную Вам функцию Write для печати на принтер, передав ей в качестве параметра идентификатор открытого канала PRT:.

Заметим, что можно написать универсальную функцию для вывода данных в канал ввода/вывода. Для этого нужно выделить какой-нибудь регистр (например, D1) для передачи ей идентификатора канала в качестве параметра.

Такая функция будет способна выводить данные на консоль, на принтер и на другие устройства.

Принтер — это устройство вывода, поэтому канал PRT: может работать только в одном направлении.

6.4.4. Работа с последовательным портом.

Работать с последовательным портом так же просто, как и с принтером. Нужно всего лишь открыть канал ввода/вывода с именем “SER:” и использовать функции Read и Write для передачи информации. Канал последовательного порта работает в обоих направлениях (на ввод и на вывод).

Параметры интерфейса порта (например, HandShake или скорость передачи) можно установить с помощью программы Preferences.

6.4.5. Синтез речи.

С помощью специальных средств операционной системы можно “заставить” Amiga разговаривать; для этого в

ОПЕРАЦИОННАЯ СИСТЕМА

базовый пакет software Amiga входит специальная программа эмуляции речевого синтезатора — "narrator.device". Работать с программно-эмулируемыми устройствами (.device) несколько сложнее, чем с рассмотренными ранее, так как такие устройства не имеют стандартных каналов ввода/вывода.

В современных версиях ОС по непонятным причинам поддержка синтезатора речи прекращена разработчиками системы, и в настоящее время система не комплектуется narrator.device и translator.library. Использовать эти функции тем не менее можно, для этого достаточно переписать вышеупомянутые файлы из комплекта старых поставок системы. Однако, их наличие в составе современной ОС не гарантируется, и это следует учитывать.

Для начала рассмотрим основные шаги по инициализации narrator.device.

Определим некоторые константы:

***** Narrator Basic Functions 3/87 S.D. ***** (6.4.5A)

OpenLib = -552

CloseLib = -414

ExecBase = 4

Open = -30 ;открыть файл

Close = -36 ;закрыть файл

mode_old = 1005 ;"старый" режим

OpenDevice = -444 ;открыть device

CloseDevice = -450 ;закрыть device

SendIO = -462 ;начать ввод/вывод

AbortIO = -480 ;закончить ввод/вывод

Translate = -30 ;перевести текст из обычного
;представления в фонетическое

Программа инициализации синтезатора выглядит

так:

init: ;инициализация narrator.device

;* Открываем библиотеку dos.library *

ОПЕРАЦИОННАЯ СИСТЕМА

```

move.l    ExecBase,A6           ;база библиотеки exec
lea       dosname(PC),A1        ;имя библиотеки dos.library
moveq     #0,D0                 ;любая версия
jsr       OpenLib(A6)           ;открываем библиотеку dos
move.l    D0,dosbase           ;сохраняем базу
beq       error1                ;ошибка

;* Открываем библиотеку translator.library *
lea       transname(PC),A1      ;"translator.library"
moveq     #0,D0
jsr       OpenLib(A6)           ;открываем translator
move.l    D0,transbase         ;сохраняем базу
beq       error2                ;ошибка!

;* Подготавливаем структуру ввода/вывода для narrator.device *
lea       talkio,A1             ;адрес IO-структуры
move.l    #nwrrep,14(A1)        ;адрес порта
move.l    #amaps,48+8(A1)       ;указатель на audio-карту
move      #4,48+12(A1)          ;размер audio-карты
move.l    #512,26(A1)           ;размер IO-структуры
move      #3,28(A1)             ;команда: запись (вывод)
move.l    #outtext,40(A1)       ;адрес буфера вывода

;* Открываем устройство narrator.device *
moveq     #0,D0
moveq     #0,D1                 ;обнуляем флаги
lea       nardevice(PC),A0      ;адрес строки
                                ;"narrator.device"
jsr       OpenDevice(A6)        ;открываем narrator
tst.l     D0                    ;ошибка?
bne       error3                ;да!

;* Открываем окно *
move.l    #consolname,D1        ;имя консоли
move.l    #mode_old,D2          ;режим
move.l    dosbase(PC),A6        ;загружаем базу dos.library
jsr       Open(A6)              ;открываем окно ввода/вывода
move.l    D0,conhandle          ;сохраняем идентификатор окна
beq       error4                ;ошибка!

    Для отображения текста, "произносимого" синтезатором,
    будем использовать функцию "pmsg":
    move.l    #intext,D2         ;входной текст
    bsr      pmsg                ;печатаем в окно
sayit:                                         ;заставляем narrator говорить

```

ОПЕРАЦИОННАЯ СИСТЕМА

;* Приводим текст к фонетическому виду, "понятному" синтезатору *

```

lea      intext(PC),A0      ;адрес текста — в A0
move.l   #outtext-intext,D0 ;размер текста
lea      outtext(PC),A1     ;адрес области вывода (для
                           ;сохранения результата
                           ;трансляции)
move.l   #512,D1            ;размер области вывода
move.l   tranbase(PC),A6    ;база библиотеки translator
jsr      Translate(A6)      ;Translate — функция
                           ;библиотеки translator!!!
                           ;* Синтезируем речь *
lea      talkio(PC),A1      ;адрес IO-структуры
move.l   #512,36(A1)        ;размер области вывода
move.l   ExecBase,A6        ;адрес exec.library
jsr      SendIO(A6)         ;выводим данные на device
    
```

По окончании программы вывод данных прекращается, поэтому перед выходом поместим вызов функции "getchr" (ожидание нажатия клавиши):

```

bstr      getchr            ;ждем нажатия клавиши
    
```

После запуска программы Вы услышите некоторую фразу, "произнесенную" компьютером (естественно, по-английски). Выйти из программы можно по нажатию клавиши <Enter>.

```

qu:                                              ;(6.4.5C)
move.l    ExecBase,A6      ;адрес exec
lea       talkio(PC),A1     ;указатель на IO-структуру
jsr       AbortIO(A6)       ;закончить вывод
move.l    conhandle(PC),D1
move.l    dosbase(PC),A6
jsr       Close(A6)         ;закрываем окно
move.l    dosbase(PC),D1
move.l    ExecBase,A6
jsr       CloseLib(A6)      ;закрываем dos.library
lea       talkio(PC),A1
jsr       CloseDevice(A6)   ;закрываем narrator.device
move.l    tranbase(PC),A1
jsr       CloseLib(A6)      ;закрываем translator.library
rts                                              ;* конец программы
    
```

Блок данных для нашей программы выглядит так:

ОПЕРАЦИОННАЯ СИСТЕМА

```
mytext:    dc.b    'This is a test text!',10,13,10,13,0
dosname:   dc.b    'dos.library',0
transname: dc.b    'translator.library',0
consolname:dc.b    'RAW:0/100/640/100/** Test window',0
nardevice: dc.b    'narrator.device',0
           align
dosbase:   dc.l    0
transbase: dc.l    0
amaps:     dc.b    3,5,10,12
           align
conhandle: dc.l    0
talkio:    blk.l   20,0
nwrrep:    blk.l   8,0
intext:    dc.b    'hello, i am the amiga talking to you',0
           align
outtext:   blk.b   512,0
```

Речевой синтезатор Amiga имеет набор параметров, изменяя которые Вы можете получать различные интересные эффекты. Параметры narrator.device задаются в специальном блоке — talkio, который имеет следующую структуру:

Смещение	Длина	Назначение
** Данные порта **		
0	L	Указатель на следующий блок
4	L	Указатель на последний блок
8	B	Тип I/O
9	B	Приоритет
10	L	Указатель на имя I/O
14	L	Указатель порта
18	W	Размер структуры
** Данные ввода/вывода **		
20	L	Указатель на device
24	L	Указатель на блок device
28	W	Команда
30	B	I/O-флаги
31	B	I/O-статус
32	L	I/O-указатель

ОПЕРАЦИОННАЯ СИСТЕМА

36	L	Размер I/O-блока
40	L	Указатель на I/O-данные
44	L	I/O-смещение
** Данные narrator.device **		
48	W	Скорость воспроизведения речи
50	W	Высота основного тона
52	W	Режим синтеза
54	W	"Пол" говорящего (М/Ж)
56	L	Указатель на audio-карту
60	W	Число элементов audio-карты
62	W	Громкость
64	W	Период
66	B	Флаг графического сопровождения (0=выключено)
67	B	Маска (для внутреннего использования)
68	B	Номер канала (для внутреннего использования)

Мы не рекомендуем экспериментировать с данными в первых двух блоках, так как это может привести к сбою или зависанию компьютера. Нас больше будут интересовать параметры narrator.device, задаваемые в последнем блоке структуры talkio.

Рассмотрим некоторые из них (числа в скобках соответствуют стандартным значениям параметров):

Скорость воспроизведения речи (150).

Этот параметр определяет насколько быстро синтезатор должен "произносить" слова. Высота "голоса" остается неизменной.

Высота основного тона (110).

Изменяя этот параметр (от 65 до 320), Вы сможете подобрать наиболее "правдоподобную" высоту синтезируемого голоса.

Режим синтеза.

Определяет "натуральность" голоса. Нулевое значение задает наиболее монотонную речь (режим "робота").

ОПЕРАЦИОННАЯ СИСТЕМА

“Пол” говорящего (0).

Нулевое значение задает режим синтеза мужского голоса, ненулевое — женского.

Громкость (64).

Этот параметр определяет громкость голоса (0-64).

Период (22200).

Уменьшая это значение, Вы сможете получать более “низкие” голоса.

И наконец, приведем полный листинг программы, работающей с речевым синтезатором:

```
;***** Speech output S.D. *****
```

```
OpenLib      = -552
```

```
CloseLib     = -414
```

```
ExecBase    = 4
```

```
;* Функции AmigaDOS:
```

```
Open        = -30
```

```
Close       = -36
```

```
OpenDevice  = -444
```

```
CloseDevice = -450
```

```
AddPort    = -354
```

```
RemPort     = -360
```

```
;DoIO       = -456
```

```
SendIO      = -462
```

```
AbortIO     = -480
```

```
Read        = -42
```

```
Write       = -48
```

```
;MyInput    = -54
```

```
;Output     = -60
```

```
;CurrDir    = -126
```

```
;Exit       = -144
```

```
WaitForChar = -204
```

```
FindTask    = -294
```

```
Translate   = -30
```

```
mode_old    = 1005
```

```
;mode_new   = 1006
```

```
;alloc_abs  = -SCC
```

```
;free_mem   = -D2
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

;!!! если > 500 К !!!
; org $40000
; load $40000
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!
ILABEL      AssemPro:includes/Amiga.l      ;только для AssemPro
INIT_AMIGA      ;только для AssemPro!
run:
        bsr      init      ;инициализация
        bra      test      ;основная программа

init:
        move.l    ExecBase,A6      ;база ехес.library — в A6
        lea        dosname(PC),A1  ;указатель на имя dos.library
        moveq     #0,D0            ;номер версии: любой
        jsr        OpenLib(A6)     ;открываем dos.library
        move.l     D0,dosbase      ;сохраняем базу
        beq        error           ;ошибка!
;*
        move.l     ExecBase,A6
        lea        transname(PC),A1
        moveq     #0,D0
        jsr        OpenLib(A6)
        move.l     D0,transbase    ;сохраняем базовый адрес
        beq        error           ;ошибка!
;*
        sub.l      A1,A1           ;начальные установки:
        move.l     ExecBase,A6     ;обнуляем A1
        jsr        FindTask(A6)    ;находим задачу
        move.l     D0,nwrrep+2     ;установка порта
        lea        nwrrep(PC),A1
        jsr        addport(A6)     ;добавляем порт
;*
        lea        talkio,A1       ;открываем narrator.device
        move.l     #nwrrep,14(A1)  ;устанавливаем адрес порта
        moveq     #0,D0
        moveq     #0,D1           ;обнуляем флаги
        lea        nardevice(PC),A0 ;имя устройства
        jsr        OpenDevice(A6)
        tst.l      D0             ;ошибка?
        bne        error           ;да!
;*
        ;инициализируем IO-блок

```

ОПЕРАЦИОННАЯ СИСТЕМА

```

lea      talkio(PC),A1
move.l   #amaps,48+8(A1) ;установка audio-карты
move.l   #4,48+12(A1)   ;размер карты
lea      consolname(PC),A1 ;определение консоли
move.l   #mode_old,D0
bsr      openfile        ;открываем окно ввода/вывода
beq      error           ;ошибка!
move.l   D0,conhandle    ;сохраняем идентификатор окна
rts

test:
move.l   #MyText,D0
bsr      pmsg            ;выводим текст на экран
bsr      sayit           ;синтезируем речь
bsr      readln          ;ввод с клавиатуры
move     #10,D0
bsr      pchar           ;выводим LF
move.l   #inline+2,D0
bsr      pmsg            ;и еще раз
bsr      prclrf
bra      qu

error:
moveq    #-1,D7          ;Флаг (для EXIT_AMIGA)

qu:
move.l   ExecBase,A6
lea      talkio,A1
jsr      AbortIO(A6)     ;прерываем вывод
move.l   conhandle(PC),D1
move.l   dosbase(PC),A6
jsr      Close(A6)       ;закрываем окно
move.l   dosbase(PC),A1
move.l   ExecBase,A6
jsr      CloseLib(A6)    ;закрываем dos.library
lea      nwrrep(PC),A1
jsr      RemPort(A6)     ;удаляем порт ввода/вывода
lea      talkio(PC),A1
jsr      CloseDev(A6)    ;закрываем narrator.device
move.l   tranbase(PC),A1
jsr      CloseLib(A6)    ;закрываем translator.library
EXIT_AMIGA              ;только для AssemPro
openfile:               ;открываем файл

```

ОПЕРАЦИОННАЯ СИСТЕМА

```

    move.l    A1,D1
    move.l    D0,D2
    move.l    dosbase(PC),A6
    jsr       Open(A6)
    tst.l     D0
    rts

pmsg:
                                ;печать сообщения по (D0)
    movem.l   D0-D7/A0-A6,-(SP)
    move.l    D0,A0
    move.l    A0,D2
    moveq     #0,D3

mess1:
    tst.b     (A0)+
    beq.s     mess2
    addq.l    #1,D3
    bra.s     mess1              ;вычисляем длину

mess2:
    move.l    conhandle(PC),D1
    move.l    dosbase(PC),A6
    jsr       Write(A6)          ;печатаем строку
    movem.l   (SP)+,D0-D7/A0-A6
    rts

pcrlf:
    move      #10,D0
    bsr       pchar
    move      #13,D0

pchar:
                                ;вывод символа по коду в D0
    movem.l   D0-D7/A0-A6,-(SP)
    move.l    conhandle(PC),D1

pch1:
    lea       chbuff(PC),A1
    move.b    D0,(A1)
    move.l    A1,D2
    moveq     #1,D3              ;выводим 1 символ
    move.l    dosbase(PC),A6
    jsr       Write(A6)
    movem.l   (SP)+,D0-D7/A0-A6;восстанавливаем регистры
    rts

scankey:
                                ;проверка нажатия клавиши
    move.l    conhandle(PC),D1

```

ОПЕРАЦИОННАЯ СИСТЕМА

```

move.l    #500,D2                ;время ожидания
move.l    dosbase(PC),A6
jsr       WaitForChar(A6)
tst.l     D0
rts

readln:
movem.l   D0-D7/A0-A6,-(SP) ;сохраняем регистры
lea        inline+2,A2          ;буфер ввода
clr.l     (A2)

inplop:
bsr       getchr
cmp.b     #8,D0
beq.s     backspace
cmp.b     #127,D0                ;"Delete"?
beq.s     backspace
bsr       pchar
cmp.b     #13,D0                 ;"Enter"?
beq.s     inputx
move.b    D0,(A2)+
bra.s     inplop

inputx:
clr.b     (A2)+
sub.l     #inline,A2
move      A2,inline             ;длина строки плюс 1
movem.l   (SP)+,D0-D7/A0-A6;восстанавливаем регистры
rts

backspace:
cmp.l     #inline,A2            ;курсор в начале строки?
beq.s     inplop                ;да, не удалять
move.b    #8,D0
bsr       pchar                 ;курсор — влево
move      #32,D0                ;пробел
bsr       pchar
move      #8,D0
bsr       pchar                 ;"забой"
clr.b     (A2)
subq.l    #1,A2
bra       inplop

getchr:
move.l    #1,D3                ;ввести один символ

```

ОПЕРАЦИОННАЯ СИСТЕМА

```

move.l    conhandle(PC),D1
lea       inbuff(PC),A1      ;буфер ввода
move.l    A1,D2
move.l    dosbase(PC),A6
jsr       Read(A6)
moveq     #0,D0
move.b    inbuff(PC),D0      ;возвращаем код введенного
                               ;символа

rts

sayit:
lea       intext(PC),A0
move.l    #outtext-intext,D0
lea       outtext(PC),A1
move.l    #512,D1
move.l    tranbase(PC),A6
jsr       Translate(A6)

p:
lea       talkio(PC),A1
move      #3,28(A1)          ;??
move.l    #512,36(A1)
move.l    #outtext,40(A1)
move.l    ExecBase,A6
jsr       SendIO(A6)
rts

MyText:   dc.b    'This is our Test-Text !',10,13,10,13,0
dosname:   dc.b    'dos.library',0
transname: dc.b    'translator.library',0
align.w
dosbase:   dc.l    0
tranbase:  dc.l    0
consolname: dc.b    'RAW:0/100/640/100/* Speech-test S.D.*',0
nardevice: dc.b    'narrator.device'
amaps:     dc.b    3,5,10,12,0,0
align.w
conhandle: dc.l    0
inbuff:    blk.b    8
inline:     blk.b    180,0
chbuff:     blk.b    82,0
narread:    blk.l    20,0
talkio:     blk.l    20,0

```

ОПЕРАЦИОННАЯ СИСТЕМА

```
nwrrep:    blk.l      8,0
intext:    dc.b       'hello, i am the amiga computer',0
           align.w
outtext:    blk.l      128,0
           end
```

6.5. Работа с дисковым.

Дисковод (винчестер), по всей видимости, является одним из наиболее важных внешних устройств Amiga. В этом пункте мы рассмотрим основные функции, предоставляемые операционной системой для работы с файлами на диске.

Начнем с некоторых базовых операций. Прежде всего, для работы с любым файлом его необходимо сначала "открыть" (подобно тому, как нужно открывать библиотеки и программно-эмулируемые устройства). Для этого библиотека dos.library содержит специальную функцию — Open, с которой мы уже встречались при работе с консолью.

6.5.1. Открытие и закрытие файлов.

Как мы уже говорили, функция Open вызывается с тремя аргументами. Среди них есть специальный параметр, задающий режим открытия файла (D2). Если требуется открыть уже существующий файл, то этот параметр должен содержать число 1005 (mode_old).

Если же Вы хотите создать новый файл, используйте режим mode_new (1006). Однако следует помнить, что если при открытии в режиме mode_new файл с таким именем уже существует, то его старое содержимое будет уничтожено! Чтобы избежать подобных случаев, Ваша программа должна проверять отсутствие одноименного файла на диске перед его созданием (проверить существование файла с данным именем можно, например, с помощью системной функции Lock, о которой речь пойдет в пункте 6.5.6).

Итак, начнем с написания подпрограммы, открывающей требуемый файл. Пусть ASCII-строка, задающая имя файла, помечена как filename (будем считать, что эта стро-

ОПЕРАЦИОННАЯ СИСТЕМА

ка заканчивается нулевым байтом). Для передачи режима открытия файла нашей подпрограмме будем использовать регистр D2.

После вызова Open мы должны сохранить возвращенный идентификатор открытого канала в специальной переменной, подобно тому, как мы сохраняли идентификатор окна (консоли) в переменной conhandle. Если после этого сразу написать rts, то вызывающая программа сможет обнаружить ошибку открытия файла по состоянию флага Z (с помощью ветвлений BEQ/VNE).

Подпрограммы открытия и закрытия файлов выглядят так:

```

Open      = -30                      ;(6.5.1A)
Close     = -36
mode_old  = 1005
mode_new  = 1006
...
openfile: move.l   dosbase(PC),A6      ;базовый адрес dos.library
          move.l   #filename,D1       ;указатель на имя файла
          jsr      Open(A6)           ;открываем файл
          move.l   D0,filehd          ;сохраняем идентификатор
                                      ;(в случае ошибки флаг Z = 1)
          rts
closefile:
          move.l   dosbase(PC),A6
          move.l   filehd(PC),D1      ;идентификатор закрываемого
                                      ;файла
          jsr      Close(A6)          ;закрываем файл
          rts
filehd:   dc.l     0                  ;переменная для хранения
                                      ;идентификатора файла
filename: dc.b     "Filename",0      ;имя открываемого файла
          align    2,even            ;или even
    
```

Операция закрытия файла используется для освобождения ресурсов системы, выделенных функцией Open. Для каждого открытого в Вашей программе файла рано или поздно должна быть вызвана функция Close.

ОПЕРАЦИОННАЯ СИСТЕМА

6.5.2. Чтение и запись данных.

Предположим, что нам нужно создать новый файл и записать в него некоторый текст. Для начала мы должны написать:

```

move.l    #mode_new,D2    ;режим mode_new
bsr       openfile        ;создаем файл
beq       error           ;реакция на ошибку
    
```

Пусть по адресу filename записана строка "Testfile". Тогда после вызова подпрограммы openfile в приведенном фрагменте на диске появится новый файл с именем "Testfile" (если файл с таким именем уже существует, то он предварительно удалится).

Текст для записи в файл определим так:

```

text:      dc.b           "This is a text for the Testfile",0
textend:
    
```

Метка textend введена для упрощения вычисления количества байт в тексте.

Запись данных в файл осуществляется с помощью уже известной Вам функции Write, которая, напомним, вызывается с тремя параметрами:

```

в D1      идентификатор канала (файла),
           полученный при открытии
в D2      указатель на записываемые данные
в D3      количество байт для записи
    
```

В нашем случае фрагмент программы для вызова Write будет выглядеть так:

```

Write      = -48          ;(6.5.2B)
...
writedata:
move.l     dosbase(PC),A6 ;запись данных в файл
move.l     filehd(PC),D1  ;базовый адрес dos.library
jsr        Write(A6)      ;записываем данные
rts        ;возврат
    
```

А в главной программе можно написать:

```

move.l     #text,D2       ;указатель на данные
move.l     #textend-text,D3 ;количество байт для записи
bsr        writedata      ;вызов подпрограммы записи
    
```

ОПЕРАЦИОННАЯ СИСТЕМА

После этого файл "Testfile" нам больше не нужен и мы используем функцию Close для освобождения ресурсов системы:

```

bsr      closefile      ;закрываем файл!
bra      end             ;конец программы

```

После запуска получившейся программы Вы увидите в текущем каталоге новый файл — "Testfile". Длина этого файла будет совпадать с длиной записанного в него текста.

Теперь считаем данные из файла и убедимся, что он действительно содержит нужный текст. Для этого, как не трудно догадаться, можно воспользоваться функцией Read, которая имеет такие же параметры, что и Write. Однако здесь есть одна особенность: если в D3 указать число, большее чем длина файла, то в буфер загрузится весь файл целиком, а в D0 возвратится количество реально прочитанных байт.

Начнем с резервирования памяти для буфера ввода:

```

field:   blk.b    100      ;резервируем 100 байт

```

Для нашего примера 100 байт — это более чем достаточно. Однако для работы с "настоящими" файлами Вам может понадобится буфер существенно большего размера. Напишем теперь подпрограмму для чтения данных из файла. Будем использовать регистр D2 для передачи ей указателя на буфер ввода:

```

Read     = -42              ;(6.5.2C)
...
readdata:
        move.l   dosbase(PC),A6 ;база dos.library — в A6
        move.l   filehd(PC),D1  ;идентификатор файла — в D1
        move.l   #100,D3         ;читаем не больше 100 байт
                                   ;(в нашем случае прочтется
                                   ;весь файл)

        jsr      Read(A6)
        rts              ;возврат

```

Использовать эту подпрограмму можно, например, так:

ОПЕРАЦИОННАЯ СИСТЕМА

```

move.l #mode_old,D2 ;"старый" режим
bsr openfile ;открываем файл
beq error ;ошибка!
move.l #field(PC),D2 ;адрес буфера ввода — в D2
bsr readdata ;вызов подпрограммы чтения
;данных из файла
move.l D0,D6 ;сохраняем число прочитанных
;байт в D6
bsr closefile ;закрываем файл
...
;и так далее

```

Протестировав этот фрагмент в отладчике (здесь мы подразумеваем, что библиотека dos.library уже открыта где-то раньше в программе), Вы увидите, что буфер ввода действительно будет содержать текст, ранее записанный в файл, а регистр D6 будет содержать длину этого текста.

6.5.3. Удаление файлов.

После проведенных выше экспериментов с файлом Testfile Вам вполне может понадобиться его удалить. Для этого библиотека dos.library содержит специальную функцию — DeleteFile, которая имеет смещение -72 относительно базового адреса библиотеки. Эта функция требует всего один параметр — указатель на имя удаляемого файла (в D1).

Чтобы удалить "Testfile", можно использовать следующий фрагмент кода:

```

DeleteFile = -72 ;(6.5.3)
...
move.l dosbase(PC),A6 ;загрузка базового адреса
;dos.library
move.l #filename,D1 ;имя удаляемого файла
jsr DeleteFile(A6) ;удаляем!

```

После этого "Testfile" исчезает из текущего каталога. Восстановить удаленный файл очень непросто, однако некоторые программы используют для этого специальный трюк, о котором мы поговорим позже.

ОПЕРАЦИОННАЯ СИСТЕМА

6.5.4. Переименование файлов.

Когда, скажем, текстовый редактор сохраняет измененный текст на диске, то старый файл обычно не стирается. Вместо этого редактор изменяет его имя (например, на "backup"), после чего без проблем записывает новый файл.

Для переименования файлов используется функция `Rename`, которая также входит в состав `dos.library`. Эта функция имеет смещение `-78` и вызывается с двумя параметрами: в `D1` передается указатель на старое имя файла, а в `D2` — на новое.

Переименовать файл "Testfile" в "Backup" можно, например, так:

```
Rename = -78
...
move.l dosbase(PC),A6      ;база dos.library
move.l #oldname,D1         ;указатель на старое имя файла
move.l #newname,D2         ;указатель на новое имя файла
jsr     Rename(A6)         ;переименовываем
...
oldname: dc.b "Testfile",0
newname: dc.b "Backup",0
```

6.5.5. Команды CLI.

Представьте, что Вы написали, к примеру, текстовый редактор и хотите добавить к нему функцию загрузки файла с диска. Согласитесь, что редко удастся с первого раза правильно ввести имя нужного файла, не зная содержимого текущего каталога.

Для решения этой проблемы можно предварительно вывести имена всех файлов на экран. Рассмотрим, как можно проще всего это сделать.

Библиотека "dos.library" содержит функцию `Execute`, которая позволяет выполнять команды интерпретатора командной строки не выходя из Вашей программы. Эта функция имеет смещение `-222` и вызывается с тремя параметрами:

ОПЕРАЦИОННАЯ СИСТЕМА

- в D1 указатель на ASCII-строку,
 содержащую имя команды для
 выполнения в формате интерпретатора
 командной строки (CLI).
- в D2 идентификатор файла ввода.
 Обычно этот параметр содержит ноль,
 однако указав идентификатор какого-
 либо текстового файла, Вы можете
 выполнять последовательности команд,
 записанные в этом файле. Более того,
 задав вместо этого идентификатор
 консоли, Вы фактически открываете
 новое CLI-окно.
- в D3 идентификатор файла вывода.
 Если этот параметр равен нулю,
 стандартный вывод выполняемых
 команд будет направлен на
 CLI-консоль.

Предположим, что Вы уже открыли библиотеку dos.library и окно ввода/вывода. Тогда для вызова функции Execute Вы можете использовать следующую подпрограмму:

```
Execute        = -222                                ;(6.5.5)
...

dir:
      move.l    dosbase(PC),A6                    ;база dos.library
      move.l    #command,D1                     ;указатель на строку команды
      moveq     #0,D2
      move.l    conhandle(PC),D3                ;вывод
      jsr       Execute(A6)                     ;выполняем команду
      rts
command:       dc.b    'dir',0                   ;команда для выполнения
```

Вместо 'dir' мы могли бы написать любую другую команду CLI (например, 'list'). Недостатком данного метода является то, что система должна иметь доступ к диску Workbench (а именно, к каталогу "c:"), так как все команды интерпретатора командной строки являются отдельными

ОПЕРАЦИОННАЯ СИСТЕМА

программами и, как правило, размещаются на системном диске (для владельцев жесткого диска (HD) это не является проблемой). Однако использование команд CLI сильно упрощает код и позволяет писать более компактные программы.

Приведем теперь полный текст программы, работающей с функцией Execute:

```
;***** 6.5.5 ADIR.ASM S.D. *****
```

```
OpenLib      = -552
CloseLib     = -414
;ExecBase    = 4
```

```
;* вызовы AmigaDOS:
```

```
Open         = -30
Close        = -36
Execute      = -222
IoErr        = -132
mode_old     = 1005
alloc_abs    = -$CC
```

```
ILABEL      AssemPro:includes/Amiga.i ;только для AssemPro
INIT_AMIGA  ;только для AssemPro
```

```
run:
```

```
    bsr.s    init           ;инициализация
    bra.s    test          ;основная часть
```

```
init:
```

```
    move.l   ExecBase,A6    ;база exec.library
    lea      dosname(PC),A1
    moveq    #0,D0
    jsr      OpenLib(A6)    ;открываем dos.library
    move.l   D0,dosbase     ;сохраняем базу DOS
    beq      error
    lea      consolname(PC),A1 ;параметры консоли
    move.l   #mode_old,D0
    bsr      openfile       ;открываем консоль
    beq      error
    move.l   d0,conhandle   ;сохраняем ID
    rts
```

```
test:
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

dir:      bsr      dir          ;читаем каталог
          bra      qu          ;выход

          move.l    dosbase(PC),A6 ;база dos.library
          move.l    #command,D1    ;команда CLI
          moveq     #0,D2          ;ввод = 0
          move.l    conhandle(PC),D3 ;вывод = консоль
          jsr       Execute(A6)    ;выполнить команду
          rts        ;возврат

error:

          moveq     #-1,D7         ;для AssemPro
          moveq     #0,D5

qu:

          move.l    conhandle(PC),D1
          move.l    dosbase(PC),A6
          jsr       Close(A6)      ;закрываем консоль
          move.l    dosbase(PC),A1
          move.l    ExecBase,A6
          jsr       CloseLib(A6)   ;закрываем dos.library
          EXIT_AMIGA              ;только для AssemPro
openfile:                                ;подпрограмма открытия
                                          ;файла
          move.l    A1,D1
          move.l    D0,D2
          move.l    dosbase(PC),A6
          jsr       Open(A6)       ;открываем файл
          tst.l     D0
          rts

dosname:   dc.b     'dos.library',0
          align.w
dosbase:   dc.l     0
consolname: dc.b    'CON:0/100/640/100/** CLI-Console **',0
          align.w
conhandle: dc.l     0
command:   dc.b     "dir",0
          end
    
```

6.5.6. Чтение каталога диска.

А сейчас рассмотрим другой метод получения списка

ОПЕРАЦИОННАЯ СИСТЕМА

файлов в текущем каталоге, который не требует постоянного доступа к системному диску.

А именно, напомним собственный аналог программы "Dir", которая использует функции операционной системы. Сначала необходимо передать системе информацию о том, с каким каталогом мы будем работать. Для этого в состав "dos.library" входит функция Lock ("захват", смещение - 84), которая вызывается с двумя параметрами:

- | | |
|------|---|
| в D1 | указатель на ASCII-строку, содержащую путь к нужному каталогу. Например, если требуется прочитать содержимое RAM-диска, нужно написать "RAM:", 0. |
| в D2 | режим работы с каталогом (чтение или запись). Мы будем использовать режим "Read" (-2). |

После вызова функции Lock регистр D0 будет содержать либо указатель на специальную служебную структуру — ключ, либо нулевое значение (ошибка — файл не найден). Возвращенный указатель нужно сохранить для дальнейшего использования.

Следующая функция, которую мы будем использовать — Examine (исследовать). Эта функция возвращает дополнительную информацию об исследуемом каталоге в специальном блоке FileInfoBlock. Перед вызовом Examine регистр D2 должен содержать адрес области памяти для размещения блока FileInfoBlock, а D1 — ключ каталога, возвращенный функцией Lock.

Область FileInfoBlock занимает 260 байт и содержит всю информацию об исследуемом файле (например, имя файла располагается с 9-го байта). В конце этого пункта мы приведем более подробное описание структуры блока FileInfoBlock.

Полученную информацию мы можем теперь использовать для чтения содержимого каталога с помощью функции ExNext (Examine Next, исследовать очередную запись).

ОПЕРАЦИОННАЯ СИСТЕМА

Эта функция возвращает информацию о текущей записи в каталоге и вызывается с такими же параметрами, что и Examine (D1 — ключ каталога, D2 — адрес блока FileInfoBlock, который к моменту вызова ExNext должен содержать информацию о предыдущей записи). Каждый последующий вызов ExNext выбирает очередную запись каталога и возвращает в D0 статус операции (если ноль, то все записи выбраны). Таким образом, мы должны вызывать ExNext и печатать полученную информацию о файлах до тех пор, пока ExNext не возвратит ноль.

При завершении работы с каталогом необходимо снять с него “захват”. Для этого используется функция Unlock (смещение -90)

Вот пример программы, которая открывает окно и печатает в нем список файлов диска (DF0:):

```
;6.5.5B.ASM ;***** DOS-Sample function 3/87 S.D. *****
```

```
OpenLib      = -552
CloseLib     = -414
ExBase       = 4
```

* Вызовы AmigaDOS

```
Open          = -30
Close         = -36
Read          = -42
Write         = -48
MyInput       = -54
Output        = -60
CurrDir       = -126
Lock          = -84
Unlock        = -90
Examine       = -102
ExNext        = -108
Exit          = -144
IoErr         = -132
WaitForChar   = -204
Mode          = 0
mode_old      = 1005
mode_new      = 1006
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

ILABEL    AssemPro:includes/Amiga.i    ;только для AssemPro
INIT_AMIGA
run:
        bsr.s    init
        bra      test

init:
        move.l    ExBase,A6
        lea       dosname(PC),A1
        moveq     #0,D0
        jsr       OpenLib(A6)          ;открываем dos.library
        move.l    D0,dosbase
        beq       error1               ;обработка ошибки
        lea       consolname(PC),A1
        move.l    #mode_old,D0
        bsr.s     openfile              ;открываем консоль
        beq       error2               ;ошибка?
        move.l    D0,conhandle         ;ok, сохраняем id
        rts

test:
        move.l    #MyText,D0
        bsr.s     pmsg                  ;вывод текста подсказки
        move.l    dosbase(PC),A6
        move.l    #name,D1
        moveq.l   #-2,D2                ;режим захвата — Shared
        jsr       Lock(A6)              ;захват каталога
        move.l    D0,D5
        tst.l     D0
        beq       error3               ;ошибка?
        move.l    d0,locksav
        move.l    dosbase(PC),A6
        move.l    locksav(PC),D1
        move.l    #fileinfo,D2
        jsr       Examine(A6)           ;Examine каталог
        move.l    D0,D6

loop:
        move.l    dosbase(PC),A6
        move.l    locksav(PC),D1
        move.l    #fileinfo,D2
        jsr       ExNext(A6)           ;Examine очередную запись
        tst.l     D0                   ;все?
    
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

    beq     end           ;да, выход
    move.l  #fileinfo+8,D0
    bsr.s   pmsg          ;выводим имя записи
    bsr.s   pcrlf
    bra.s   loop          ;цикл

end:
    move.l  #presskey,d0
    bsr.s   pmsg
    bsr.s   getchrl      ;ожидаем нажатия клавиши
    move.l  locksav(PC),D1
    jsr     Unlock(A6)    ;освобождаем захват

error3:
    move.l  conhandle,D1
    move.l  dosbase(PC),A6
    jsr     Close(A6)     ;закрываем консоль

error2:
    move.l  dosbase(PC),A1
    move.l  ExBase,A6
    jsr     CloseLib(A6)  ;закрываем dos.library

error1:
    EXIT_AMIGA           ;все!

openfile:
                                ;открываем файл
    move.l  A1,D1
    move.l  D0,D2
    move.l  dosbase(PC),A6
    jsr     Open(A6)
    lsl.l   D0
    rts

pmsg:
                                ;печать сообщения по (D0)
    movem.l D0-D7/A0-A6,-(SP)
    move.l  D0,A0
    move.l  A0,D2
    moveq   #0,D3

mess1:
    tst.b   (A0)+
    beq.s   mess2
    addq.l  #1,D3
    bra.s   mess1         ;вычисляем длину

mess2:
    move.l  conhandle(PC),D1

```

ОПЕРАЦИОННАЯ СИСТЕМА

```
        move.l    dosbase(PC),A6
        jsr       Write(A6)           ;печатаем строку
        movem.l   (SP)+,D0-D7/A0-A6
        rts

pcrlf:
        move      #10,D0
        bsr       pchar
        move      #13,D0

pchar:
                                   ;вывод символа по коду в D0
        movem.l   D0-D7/A0-A6,-(SP)
        move.l     conhandle(PC),D1

pch1:
        lea        chbuff(PC),A1
        move.b     D0,(A1)
        move.l     A1,D2
        moveq.l    #1,D3             ;выводим 1 символ
        move.l     dosbase(PC),A6
        jsr        Write(A6)
        movem.l   (SP)+,D0-D7/A0-A6;восстанавливаем регистры
        rts

scankey:
                                   ;проверка нажатия клавиши
        move.l     conhandle(PC),D1
        move.l     #500,D2           ;время ожидания
        move.l     dosbase(PC),A6
        jsr        WaitForChar(A6)
        tst.l      D0
        rts

readln:
        movem.l   D0-D7/A0-A6,-(SP);сохраняем регистры
        lea        inline+2,A2       ;буфер ввода
        clr.l      (A2)

inplp:
        bsr       getchr
        cmp.b     #8,D0
        beq.s     backspace
        cmp.b     #127,D0           ;"Delete"?
        beq.s     backspace
        bsr       pchar
        cmp.b     #13,D0            ;"Enter"?
        beq.s     inputx
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

        move.b    D0,(A2)+
        bra.s     inplop
inputx:
        clr.b     (A2)+
        sub.l     #inline,A2
        move      A2,inline           ;длина строки плюс 1
        movem.l   (SP)+,D0-D7/A0-A6;восстанавливаем регистры
        rts
backspace:
        cmp.l     #inline,A2           ;курсор в начале строки?
        beq.s     inplop               ;да, не удалять
        move.b    #8,D0
        bsr       pchar                ;курсор — влево
        move      #32,D0               ;пробел
        bsr       pchar
        move      #8,D0
        bsr       pchar                ;"забой"
        clr.b     (A2)
        subq.l    #1,A2
        bra       inplop
getchr:
                                   ;ввести один символ
        move.l    #1,D3
        move.l    conhandle(PC),D1
        lea       inbuff(PC),A1       ;буфер ввода
        move.l    A1,D2
        move.l    dosbase(PC),A6
        jsr       Read(A6)
        moveq     #0,D0
        move.b    inbuff(PC),D0       ;возвращаем код введенного
                                   ;символа
        rts
MyText:  dc.b     'Directory of diskette: DF0:',10,13,10,13,0
dosname: dc.b     'dos.library',0
presskey: dc.b    'Press the RETURN key!!',0
        align.w
dosbase: dc.l     0
console: dc.b     'CON:0/100/640/100/** Directory-Test **',0
name:    dc.b     'DF0:',0
        align.w
locksav: dc.l     0
    
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

fileinfo:    ds.l    20                ;или dcb.l 20,0
conhandle:   dc.l    0
inbuff:      ds.b    8                ;или dcb.b 8,0
inline:      ds.b    180              ;или dcb.b 180,0
chbuff:      ds.b    82              ;или dcb.b 82,0
end
    
```

Наконец, приведем структуру блока FileInfoBlock:

Смещение	Имя	Назначение
0	DiskKey.L	ключ каталога (диска)
4	DirEntryType.L	тип записи (+ = каталог, - = файл)
8	FileName	имя файла (108 байт)
116	Protection.L	флаг защиты файла
120	EntryType.L	тип записи
124	Size.L	размер файла (в байтах)
128	NumBlocks.L	количество блоков, занимаемых файлом
132	Days.L	дата создания файла (день)
136	Minute.L	время создания файла (минуты)
140	Tick.L	время создания файла
144	Comment	комментарий (116 байт)

Теперь Вы можете модифицировать приведенную программу, например, для вывода размера файла. Для этого можно воспользоваться приведенной в разделе 4 подпрограммой перевода чисел в ASCII-представление, передав ей в качестве параметра значение поля размера (например, с помощью `move.l fileinfo+124,D0`)

Информация, приводимая в этом разделе, несомненно, весьма полезна, однако на практике для обеспечения файлового интерфейса обычно используют специальные библиотеки (например, `reqtools.library` by Nico Francois). Такие библиотеки предоставляют все необходимые функции для работы с файл-реквестерами и имеют удобный интерфейс взаимодействия с программами пользователя. Информацию о том, как программировать реквестеры `reqtools`, Вы може-

ОПЕРАЦИОННАЯ СИСТЕМА

те найти в документации, входящей в комплект reqtools.library.

6.5.7. Непосредственный доступ к диску.

Библиотека dos.library предоставляет все функции для работы с файлами, однако иногда требуется обращаться к диску напрямую, минуя файловую систему (например, при написании “нефайловых” игр или демонстраций). Для этого используется специальное программно-эмулируемое устройство — trackdisk.device, которое обеспечивает доступ к диску на низком уровне и позволяет работать с отдельными секторами.

Однако помните, что различные эксперименты с trackdisk.device могут привести к нежелательной модификации диска, поэтому мы рекомендуем при изучении этого пункта использовать специально выделенный тестовый диск.

Итак, начнем с инициализации trackdisk.device:

```
;** Direct Disk Access via trackdisk.device ** (6.5.6)
```

```
OpenLib      = -552
CloseLib      = -414
ExecBase      = 4
Open          = -30
Close         = -36
OpenDevice    = -444
CloseDevice   = -450
SendIO        = -462
Read          = -42
Write         = -48
WaitForChar   = -204
mode_old      = 1005
```

run:

```
    bsr.s      init                ;инициализация
    bra.s      test                ;тест init:
    move.l     ExecBase,A6
    lea        dosname(PC),A1
    moveq      #0,D0
    jsr        OpenLib(A6)         ;открываем dos.library
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

move.l    D0,dosbase
beq       error           ;ошибка!
lea       diskio(PC),A1   ;область diskio
move.l    diskrep,14(A1)  ;порт
moveq     #0,d0           ;работаем с DF0:
moveq     #0,d1           ;сбрасываем флаги
lea       trddevice(PC),A0 ;имя устройства
jsr       OpenDevice(A6)  ;открываем trackdisk.device
tst.l     D0              ;ошибка?
bne       error           ;да!
move.l    #consolname,D1
move.l    #mode_old,D2
move.l    dosbase(PC),A6
jsr       Open(A6)        ;открываем консоль
tst.l     D0
beq       error           ;ошибка!
move.l    D0,conhandle    ;сохраняем id консоли
rts       ;возврат

test:     ;место для тестовой программы
          ...             ;основная программа ...
error:    ;завершающие действия
          moveq     #-1,D7 ;флаг ошибки (для SEKA)

qu:
          move.l    execbase,A6
          lea       diskio(PC),A1
          move.l    32(A1),D7 ;IO_ACTUAL — в D7 (для
                               ;тестирования)
          move      #9,28(A1) ;команда: управление приводом
          clr.l     36(A1)    ;0 = остановить вращение диска
                               ;(floppy)
                               ;выполняем
          jsr       sendio(A6)
          move.l    conhandle(PC),D1
          move.l    dosbase(PC),A6
          jsr       Close(A6) ;закрываем консоль
          move.l    dosbase(PC),D1
          move.l    execbase,A6
          jsr       CloseLib(A6) ;закрываем dos.library
          lea       diskio(PC),A1
          jsr       CloseDevice(A6) ;закрываем trackdisk.device
          rts

```

ОПЕРАЦИОННАЯ СИСТЕМА

Чтобы можно было наблюдать результаты работы нашей программы, добавим функцию getchr:

```
getchr:                                ;ожидание нажатия клавиши
      moveq    #1,D3                    ;читаем 1 символ
      move.l   conhandle,D1             ;идентификатор окна (консоли)
      move.l   #inbuff,D2              ;адрес буфера
      move.l   dosbase,A6
      jsr      read(A6)                ;читаем клавиатуру
      rts                                           ;все!
```

Завершается текст программы, как обычно, блоком переменных и констант:

```
dosname:   dc.b   'dos.library',0
           align
consolname: dc.b   'RAW:0/100/640/50/** Wait Window',0
           align
trddevice: dc.b   'trackdisk.device',0
           align
dosbase:   dc.l   0                    ;базовый адрес dos.library - здесь
conhandle: dc.l   0                    ;идентификатор консоли
inbuff:    blk.b  80,0                ;буфер клавиатуры
diskio:    blk.l  20,0                ;I/O - структура
diskrep:   blk.l  8,0                 ;I/O - порт
diskbuff:  blk.b  512*2,0             ;место для двух секторов диска
```

Теперь приступим к написанию основной части программы, работающей с trackdisk.device. Начнем с самой простой команды — с команды управления приводом дисковод, которую мы уже использовали выше. Номер команды (в нашем случае 9) указывается в специальном поле структуры ввода/вывода (28-й и 29-й байты).

Параметром команды управления приводом является длинное слово (смещение 36), которое определяет конкретное действие (0 — выключить, 1 — включить).

Приведем список основных команд trackdisk.device:

No.	Название	Назначение
2	READ	чтение одного или нескольких секторов
3	WRITE	запись секторов
4	UPDATE	обновление буфера трека
5	CLEAR	очистка буфера трека

ОПЕРАЦИОННАЯ СИСТЕМА

9	MOTOR	включение/выключение привода дискового
10	SEEK	поиск трека
11	FORMAT	форматирование трека
12	REMOVE	команда инициализации; выполняется при смене диска
13	CHANGENUM	запрос числа перестановок диска
14	CHANGESTATE	проверка наличия диска в дисковом
15	PROTSTATUS	проверка флага защиты от записи

Рассмотрим подробнее последние три команды. Эти команды используются для различного рода проверок состояния диска и возвращают результат в специальном поле (смещение 32) структуры ввода/вывода. В приведенном выше примере мы сохраняем значение этого поля в D7, что дает возможность наблюдать результаты работы команд trackdisk.device в окне AssemPro.

Вот образец программы для передачи рассмотренных команд устройству trackdisk.device:

```
test:                                ;(6.5.6B)
    lea    diskio(PC),A1             ;адрес структуры ввода/вывода
    move   #13,28(A1)                ;номер команды (к примеру, 13)
    move.l execbase,A6               ;база exec.library
    jsr    SendIO(A6)                ;посылаем команду
```

Команда CHANGENUM (код 13) возвращает количество "манипуляций" с диском: если Вы удалите диск из дискового и затем вставите его обратно, то значение, возвращаемое командой CHANGENUM, увеличится на два (!).

Команда CHANGESTATE (код 14) возвращает нулевое значение, если в дисковом присутствует диск. Иначе возвращается \$FF.

Если диск не защищен от записи, то команда PROTSTATUS (код 15) возвращает ноль. В противном случае возвращается значение \$FF.

Теперь поговорим о командах READ и WRITE. Для этих команд требуются следующие параметры: адрес бу-

ОПЕРАЦИОННАЯ СИСТЕМА

фера ввода/вывода, количество байтов для чтения/записи, а также начальное смещение на диске.

Поскольку trackdisk.device работает с целыми секторами (размер сектора = 512 байт), количество байтов для чтения/записи должно быть кратно числу 512.

Начальное смещение на диске задается в байтах. Таким образом, для первого сектора этот параметр должен быть равен нулю, для второго — 512, и так далее:

Смещение = (Номер_Сектора — 1) * 512.

Вот пример программы, которая считывает первые два сектора диска в некоторый буфер:

```
test:                                ;(6.4.6C)
    lea     diskio(PC),A1
    move    #2,28(A1)               ;команда READ
    move.l   #diskbuff,40(A1)       ;адрес буфера
    move.l   #2*512,36(A1)           ;количество байт для чтения
    move.l   #0*512,44(A1)           ;начальное смещение на диске
    move.l   execbase,A6             ;базовый адрес exec.library
    jsr     SendIO(A6)               ;посылаем команду на trackdisk
```

Протестируйте этот фрагмент в отладчике, наблюдая за содержимым буфера diskbuff. После выполнения SendIO в этот буфер считывается с диска некоторая служебная информация. Изменив параметр начального смещения (например, на 700*512), Вы сможете наблюдать содержимое секторов данных.

Для записи данных на диск используется команда WRITE, которая вызывается с такими же параметрами, что и READ. Заметим, что при вызове WRITE данные попадают на диск не сразу, так как trackdisk.device использует буферизацию по записи. Реальная модификация трека происходит при переходе на другой трек или при выполнении специальной команды — UPDATE (код 4).

Команда FORMAT (код 11) также весьма интересна. Эта команда используется для форматирования (начальной разметки) треков с одновременной записью в них некоторых данных. При этом данные для инициализации треков

ОПЕРАЦИОННАЯ СИСТЕМА

берутся из специального буфера размером $11 \times 512 = 5632$ байт (размер одного трека). Заметим, что параметр начального смещения на диске должен быть кратен числу 5632, так как FORMAT работает с целыми треками.

Используя команду FORMAT Вы легко можете написать, например, программу копирования дискет: для копирования какого-либо трека достаточно считать его содержимое с исходного диска и затем с помощью команды FORMAT выполнить разметку и запись данных в соответствующий трек диска-копии.

Команда SEEK (код 10) используется для позиционирования головки дисководов на нужный трек без выполнения каких-либо операций чтения/записи. Единственным параметром этой команды является начальное смещение.

И, наконец, команда REMOVE (код 12) используется для установки обработчика прерываний, который вызывает-ся всякий раз, когда диск удаляется из дисководов. Адрес структуры прерываний передается в поле данных структуры ввода/вывода (смещение 40). Если этот параметр равен нулю, обработка прерывания отключается.

Приведем пример законченной программы, работающей с диском на низком уровне:

;***** Track disk — Basic function I/O, 86, S.D *****

```
ILABEL ASSEMPRO:Includes/amiga.i ;только для AssemPro
OpenLib  = -552
CloseLib  = -414
ExecBase = 4
```

*** Вызовы AmigaDOS**

```
Open      = -30
Close     = -36
OpenDevice = -444
CloseDevice = -450
SendIO    = -462
Read      = -42
Write     = -48
WaitForChar = -204
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

mode_old      = 1005
INIT_AMIGA                                         ;только для AssemPro
run:
        bsr.s   init                             ;инициализация
        bra     test                             ;основная программа

init:
        move.l   ExecBase,A6
        lea      dosname(PC),A1
        moveq    #0,D0
        jsr      OpenLib(A6)                     ;открываем dos.library
        move.l   D0,dosbase
        beq      error1                          ;ошибка! нет dos.library???
        lea      diskio(PC),A1
        move.l   #diskrep,14(A1)                 ;порт ввода/вывода
        moveq    #0,D0                           ;номер устройства
        moveq    #0,D1                           ;флаги
        lea      trdddevice(PC),A0               ;имя "trackdisk.device"
        jsr      OpenDevice(A6)                  ;открываем trackdisk
        tst.l    D0                              ;ошибка?
        bne      error2                          ;да! странно...

bp:
        lea      consolname(PC),A1
        move.l   #mode_old,D0
        bsr.s    openfile                        ;открываем окно (консоль)
        beq      error3                          ;ошибка!
        move.l   D0,conhandle                    ;сохраняем идентификатор
        rts

test:
        bsr.s    accdisk                         ;работаем с trackdisk
        bsr.s    getchr                          ;ждем нажатия клавиши
        bra.s    qu                              ;выход

qu:
        move.l   ExecBase,A6
        lea      diskio(PC),A1                   ;загружаем адрес I/O-структуры
        move     #9,28(A1)                       ;команда MOTOR
        move.l   #0,36(A1)                       ;выключить привод
        jsr      SendIO(A6)
        move.l   conhandle(PC),D1
        move.l   dosbase(PC),A6
        jsr      Close(A6)                       ;закрываем консоль error3:
    
```

ОПЕРАЦИОННАЯ СИСТЕМА

```
        lea      diskio(PC),A1
        move.l   32(A1),D7
        jsr      CloseDevice(A6)    ;закрываем trackdisk.device

error2:
        move.l   dosbase(PC),A1
        move.l   ExecBase,A6
        jsr      CloseLib(A6)      ;закрываем dos.library

error1:
        EXIT_AMIGA

openfile:
                                           ;открыть файл
        move.l   A1,D1
        move.l   D0,D2
        jsr      Open(A6)
        tst.l    D0
        rts

scankey:
                                           ;проверка нажатия клавиши
        move.l   conhandle(PC),D1
        move.l   #500,D2           ;время ожидания
        move.l   dosbase(PC),A6
        jsr      WaitForChar(A6)
        tst.l    D0
        rts

getchr:
                                           ;чтение символа с клавиатуры
        moveq.l   #1,D3
        move.l   conhandle,D1
        lea      inbuff(PC),A1    ;буфер клавиатуры
        move.l   A1,D2
        move.l   dosbase(PC),A6
        jsr      Read(A6)
        moveq     #0,D0
        move.b    inbuff(PC),D0
        rts

accdisk:
        lea      diskio(PC),A1
        move      #2,28(A1)        ;команда: READ
        move.l    #diskbuff,40(A1) ;адрес буфера чтения
        move.l    #2*512,36(A1)    ;читаем два сектора
        move.l    #20*512,44(A1)   ;смещение: 20 секторов
        move.l    ExecBase,A6
        jsr      SendIO(A6)        ;посылаем команду
```

ОПЕРАЦИОННАЯ СИСТЕМА

```

                                rts                                ;все!
dosname:   dc.b                'dos.library',0
           align.w
dosbase:   dc.l                0
console:   dc.b                'RAW:0/100/640/100/** Test-Window S.D. V0.1',0
trddevice: dc.b                'trackdisk.device',0
           align.w
conhandle: dc.l                0
inbuff:    ds.b                8,0
diskio:     ds.l               20,0
diskrep:    ds.l                8,0
diskbuff:   ds.b               512*2,0
end
```

РАБОТА С INTUITION

Раздел 7. РАБОТА С INTUITION.

Переходим к рассмотрению функций Intuition — специальной компоненты операционной системы, которая отвечает за GUI (Graphics User Interface, графический интерфейс пользователя). Именно Intuition предоставляет пользовательским программам средства для работы с экранами, окнами, меню и другими элементами графического интерфейса.

Возможности Intuition настолько широки, что используя эту систему можно написать эффективный и удобный интерфейс практически для любого приложения. Правда, за эту гибкость приходится “платить” более сложным кодом, так как даже для работы с простыми элементами интерфейса в Intuition используются громоздкие структуры с множеством параметров. Однако это не повод для беспокойства: используя некоторые заранее заготовленные таблицы и фрагменты кода Вы без проблем сможете запрограммировать собственный интерфейс.

Сразу заметим, что на задание неправильных параметров Intuition может отреагировать самым нежелательным образом (например, “завесить” компьютер), поэтому не забывайте перед запуском программ сохранять исходные тексты на диске.

Итак, приступим. Перед началом работы с Intuition необходимо открыть специальную библиотеку — “intuition.library”. Для этого можно воспользоваться следующей подпрограммой:

```
OpenLib = -552  
ExecBase = 4  
run:
```


РАБОТА С INTUITION

```

        bsr      openint
        ...
openint:                                     ;* открыть и инициализировать
                                           ;* Intuition

        move.l   ExecBase,A6
        lea      IntName(PC),A1 ;загружаем имя intuition.library
        moveq    #0,D0          ;версия — любая
        jsr      OpenLib(A6)    ;открываем библиотеку
        move.l   D0,intbase     ;сохраняем базовый адрес
        rts

IntName: dc.b    'intuition.library'.0
        align.w

intbase: dc.l    0

        По окончании работы с Intuition необходимо за-
        крыть intuition.library:
        CloseLib = -414

        ...
closeint:                                     ;* закрыть Intuition

        move.l   ExecBase,A6
        move.l   intbase(PC),A1 ;базовый адрес intuition.library
        jsr      CloseLib(A6)   ;закрываем библиотеку
        rts
    
```

7.1. Работа с экранами (screens).

Экран, по всей видимости, является основным элементом графического интерфейса операционной системы Amiga (не путайте с экраном дисплея: на дисплее могут частично отображаться несколько экранов Intuition, либо один экран целиком. Такая организация видео-ресурсов называется системой виртуальных экранов). Каждая задача может открыть один или несколько экранов, в которых могут создаваться окна и меню, а также осуществляться ввод/вывод. Отдельные экраны являются абсолютно независимыми друг от друга.

Прикладная программа и пользователь могут манипулировать экранами (например, выносить какой-либо экран на передний план, переключать экраны программы итд.).

Первая функция, которую мы рассмотрим в этом

РАБОТА С INTUITION

пункте — OpenScreen (открыть экран). Эта функция входит в состав intuition.library и используется для создания новых экранов. Параметры создаваемого экрана передаются в специальной таблице, которая состоит из 13 элементов. Заголовок этой таблицы может выглядеть, например, так:

align.w

screen_defs: ;* начало таблицы параметров экрана

Первое, что нужно определить для создания нового экрана, это его положение и размеры. Пусть, например, нам нужно создать экран, целиком занимающий видео-область дисплея. Такой экран должен иметь размер 320x200 точек (в режиме PAL), а также координаты левого верхнего угла, равные нулю: ?

```
x_pos:    dc.w    0           ;X-координата экрана
y_pos:    dc.w    0           ;Y-координата экрана
width:    dc.w    320        ;ширина экрана
height:   dc.w    200        ;высота экрана
```

После этого нужно определить цвета, которые должны использоваться при отображении графической информации на экране. Количество цветов зависит от "глубины" (числа битпланов) экрана: будем использовать четыре битплана, что позволит нам задать $2^4 = 16$ различных цветов.

```
depth:    dc.w    4           ;количество битпланов экрана
```

Следующие две записи определяют цвета заголовка, специальных символов и фона:

```
detail_pen: dc.b    0           ;цвет текста, итд.
block_pen:  dc.b    1           ;цвет заднего плана (фона)
```

Эти значения задают номера цветов в предопределенной таблице — палитре. Стандартная палитра содержит следующие цвета:

Номер	Цвет	Номер	Цвет
Background 0	темно-синий	для трех битпланов:	
1	белый	4	голубой
для двух битпланов:		5	фиолетовый
2	черный	6	бирюзовый
3	красный	7	белый

РАБОТА С INTUITION

Номер	Цвет
для четырех битпланов:	
8	черный
9	красный
10	зеленый
11	коричневый
12	синий
13	синий
14	зеленый
15	зеленый

Начиная с системы 2.0, стандартная палитра несколько изменилась — теперь первые четыре цвета выглядят соответственно как серый, черный, белый и синий. Следующая запись таблицы параметров экрана задает видео-режим и имеет следующий формат:

Бит	Маска	Имя	Назначение
1	2	GENLOCK_VIDEO	
2	4	INTERLACE	включает режим interlaced, который позволяет удвоить вертикальное разрешение
6	\$40	PFBA	
7	\$80	EXTRA_HALFBRIGH	включает 64-цветный режим EMB
8	\$100	GENLOCK_AUDIO	
10	\$400	DPLF	включает режим double playfield
11	\$800	HOLDNMODIFY	включает режим hold and modify (HAM)
13	\$2000	VP_HIDE	
14	\$4000	<u>SPRITES</u>	<u>позволяет использовать спрайты на экране</u>
15	\$8000	HIRES	включает режим высокого разрешения (640 точек по горизонтали)

Для нашего примера выберем значение 2:

view_modes: dc.w 2 ;режим экрана

Следующее слово таблицы определяет тип экрана. Значение 15 задает полностью “настраиваемый” экран (то есть экран, который использует все возможные опции).

screen_type: dc.w 15 ;тип экрана

Далее следует указатель на таблицу символов (шрифт), которая должна использоваться при выводе текстовой информации на экран. Нулевое значение задает

РАБОТА С INTUITION

стандартный шрифт.

```
font:      dc.l      0      ;шрифт — стандартный
```

Следующее поле таблицы параметров должно содержать адрес ASCII-строки, определяющей имя (заголовок) экрана:

```
title:     dc.l      sname   ;имя экрана
```

Следующая запись определяет список нестандартных символов управления экраном (gadgets). Будем использовать только стандартные символы, такие, как "Bring Forward" (вынести на передний план), итд., для чего поместим сюда нулевое значение:

```
gadgets:   dc.l      0      ;не использовать специальных
                                ;управляющих символов (gadgets)
```

И, наконец, последняя запись таблицы параметров экрана определяет битовую карту (bitmap), которая должна использоваться для работы с экраном. Как обычно, нулевое значение задает стандартную карту:

```
bitmap:    dc.l      0      ;использовать стандартный bitmap
```

Осталось только определить имя экрана:

```
sname:     dc.b      'Our Screen',0   ;заголовок экрана
```

Объединив все рассмотренные параметры, получим следующую таблицу:

```
align.w
screen_defs:                                ;* таблица параметров экрана
x_pos:     dc.w      0      ;X-координата экрана
y_pos:     dc.w      0      ;Y-координата экрана
width:     dc.w      320    ;ширина (размер по X)
height:    dc.w      200    ;высота (размер по Y)
depth:     dc.w      4      ;число битпланов
detail_pen: dc.b      0      ;цвет заголовка
block_pen:  dc.b      1      ;цвет фона
view_modes: dc.w      2      ;видео-режимы
screen_type: dc.w      15    ;тип экрана
font:      dc.l      0      ;шрифт
title:     dc.l      sname   ;указатель на имя (заголовок) экрана
gadgets:   dc.l      0      ;список управляющих символов
bitmap:    dc.l      0      ;битовая карта экрана
sname:     dc.b      'Our Screen',0 ;ASCII-строка — имя экрана
```

РАБОТА С INTUITION

После определения блока параметров можно воспользоваться функцией OpenScreen (смещение -198 в intuition.library), которая, собственно, и создаст нужный экран. Единственным параметром этой функции является адрес вышеописанной таблицы:

OpenScreen = 198

```

        bsr      openint          ;открыть Intuition
        bsr      scropen         ;открыть экран
        ...
scropen:                                ;* открыть экран
        move.l   intbase(PC),A6    ;база intuition.library
        lea      screen_defs(PC),A0 ;указатель на таблицу
                                   ;параметров
        jsr      OpenScreen(A6)    ;открываем экран
        move.l   D0,screenhd       ;сохраняем идентификатор
        rts                                     ;все!
        ...
screen_defs:                            ;начало таблицы параметров

```

После выполнения OpenScreen на дисплее появится новый экран, с которым можно работать так же, как и с любым другим экраном Intuition. Как только экран становится ненужным, его необходимо закрыть. Для этого используется функция CloseScreen (смещение -66 в intuition.library), единственным параметром которой является идентификатор закрываемого экрана:

CloseScreen = -66

```

        ...
scrclose:                                ;* закрыть экран
        move.l   intbase(PC),A6    ;база intuition.library
        move.l   screenhd(PC),A0   ;параметр для CloseScreen
        jsr      CloseScreen(A6)   ;закрываем экран
        rts                                     ;все!

```

Рассмотрим подробнее, что представляет из себя идентификатор экрана, возвращаемый функцией OpenScreen. На самом деле этот идентификатор является указателем на специальный блок данных, содержащий всю информацию о созданном экране.

Этот блок данных имеет достаточно сложную струк-

РАБОТА С INTUITION

туру и содержит в основном служебную информацию, однако некоторые поля могут представлять интерес:

Номер	Имя	Назначение
0	(NextScreen.L)	указатель на следующий экран
4	(FirstWindow.L)	указатель на первое окно
8	(LeftEdge.W)	координаты экрана
\$A	(TopEdge.W)	
\$C	(Width.W)	размеры экрана
\$E	(Height.W)	
\$10	(MouseY.W)	положение указателя мыши на экране
\$12	(MouseX.W)	
\$14	(Flags.W)	флаги
\$16	(Title.L)	указатель на текст заголовка
\$1A	(DefaultTitle)	заголовок по умолчанию
\$28	(Font.L)	указатель на таблицу символов (шрифт)
\$C0	(Plane0.L)	указатель на битплан 0
\$C4	(Plane1.L)	указатель на битплан 1
\$C8	(Plane2.L)	указатель на битплан 2
\$CC	(Plane3.L)	указатель на битплан 3

Указатели битпланов могут потребоваться, например, при написании собственной подпрограммы вывода символов. Занести адрес нужного битплана в один из адресных регистров можно так:

```
move.l    screenhd(PC),A5      ;адрес блока данных экрана
move.l    $C0(A5),A5          ;адрес нулевого битплана
```

Напишем фрагмент программы, использующей полученный адрес:

```
move.l    screenhd(PC),A5      ;адрес блока данных экрана
move.l    $C0(A5),A5          ;адрес нулевого битплана
move      #S20,D0              ;счетчик цикла — в D0

loop1:
move      #SAAAA,(A5)          ;рисует на экране
add.l     #80,A5                ;следующая строка
dbra      D0,loop1              ;цикл
```

Программа рисует столбец шириной в 16 точек, заполненный маской \$AAAA. Конечно, практической пользы от этой программы мало, однако она демонстрирует простоту работы с видео-памятью, связанной с экраном Intuition.

РАБОТА С INTUITION

О правомерности прямой работы с видеопамятью рассказано в разделе 9.

Как обычно, Вы можете перемещать экраны Intuition, "захватив" верхнюю часть (заголовок) нужного экрана с помощью мыши, однако то же самое можно делать и программно. Для этого в состав intuition.library входит функция MoveScreen (смещение -162), которая вызывается с тремя параметрами:

- в A0 идентификатор экрана, полученный при вызове OpenScreen
- в D1 смещение по вертикали (в пикселах)
- в D0 смещение по горизонтали (в пикселах), в старых версиях Intuition — не используется

Пример:

MoveScreen = -162

```
...
scrmove:                                ;* сдвигает экран на D1 пикселей вниз
        move.l    intbase(PC),A6        ;база intuition.library
        move.l    screenhd(PC),A0       ;идентификатор экрана
        moveq     #0,D0
        jsr       MoveScreen(A6)       ;сдвигаем экран
        rts
```

Напишем программу, которая использует джойстик для перемещения своего экрана (как Вы уже знаете, информация о положении джойстика доступна по адресу \$DFF00C (см. п. 5.3)). Определим для начала основные действия программы:

1. Открыть библиотеку intuition.library
2. Открыть экран
3. Обрабатывать джойстик и соответствующим образом перемещать экран
4. Закрыть экран
5. Закрыть библиотеку intuition.library

И, наконец, приведем полный листинг самой программы:

```
;** Demo program to open and move a screen (7.1) **
MoveScreen    = -162
```

РАБОТА С INTUITION

```

OpenScreen      = -198
CloseScreen     = -66
CloseLibrary    = -414
OpenLibrary     = -552
ExecBase        = 4
joy2             = $DFF00C      ;адрес порта #2 (джойстик)
fire            = $BFE001      ;кнопка: бит 7
run:
        bsr      openint        ;открываем intuition.library
        bsr      scropen       ;открываем экран
        move     joy2,D6        ;сохраняем состояние джойстика
loop:
        tst.b    fire           ;проверяем нажатие кнопки
        bpl.s    ende          ;нажата = выход
        move     joy2,D0        ;текущее состояние джойстика
        sub      D6,D0
        cmp      #$0100,D0     ;вверх?
        bne.s    noup         ;нет...
        moveq    #-1,D1        ;иначе смещение по Y = -1
        bsr.s    scrmove       ;сдвигаем экран
        bra.s    loop         ;цикл
noup:
        cmp      #$0001,D0     ;вниз?
        bne.s    loop         ;нет, цикл
        moveq    #1,D1        ;иначе смещение по Y = 1
        bsr.s    scrmove       ;сдвигаем экран
ende:
        bsr.s    scrclose      ;закрываем экран
        bsr.s    closeint      ;закрываем intuition.library
        rts                  ;все!
openint:
        ;* открыть и инициализировать
        ;* Intuition
        move.l    ExecBase,A6
        lea       IntName(PC),A1 ;загружаем имя intuition.library
        moveq     #0,D0        ;версия — любая
        jsr       OpenLib(A6)  ;открываем библиотеку
        move.l     D0,intbase   ;сохраняем базовый адрес
        rts
closeint:
        ;* закрыть Intuition
        move.l     ExecBase,A6
    
```


РАБОТА С INTUITION

```

move.l    intbase(PC),A1    ;базовый адрес intuition.library
jsr       CloseLib(A6)      ;закрываем библиотеку
rts

scropen:                                     ;* открыть экран
move.l    intbase(PC),A6
lea       screen_defs(PC),A0 ;параметры экрана
jsr       OpenScreen(A6)    ;открываем экран
move.l    D0,screenhd       ;сохраняем идентификатор
rts

scrclose:                                     ;* закрыть экран
move.l    intbase(PC),A6
move.l    screenhd(PC),A0   ;идентификатор экрана — в A0
jsr       CloseScreen(A6)   ;закрываем экран
rts

scrmove:                                     ;передвинуть экран
move.l    intbase(PC),A6
move.l    screenhd(PC),A0   ;идентификатор экрана — в A0
moveq.l   #0,D0             ;горизонтальное смещение = 0
jsr       MoveScreen(A6)    ;двигаем экран
rts                               ;все
screen_defs: align 4         ;* начало таблицы параметров
                                     ;экрана
x_pos:    dc.w    0          ;X-координата экрана
y_pos:    dc.w    0          ;Y-координата экрана
width:    dc.w    320        ;ширина (размер по X)
height:   dc.w    200        ;высота (размер по Y)
depth:    dc.w    4          ;число битпланов
detail_pen: dc.b    0        ;цвет заголовка
block_pen: dc.b    1         ;цвет фона
view_modes: dc.w    2        ;видео-режимы
screen_type: dc.w    15      ;тип экрана
font:     dc.l    0          ;шрифт
title:    dc.l    sname      ;указатель на имя (заголовок)
                                     ;экрана
gadgets:  dc.l    0          ;список управляющих символов
bitmap:   dc.l    0          ;битовая карта экрана
sname:    dc.b    ;'Our Screen',0 ;ASCII-строка — имя экрана
;name:    align.w    intuition.library,0
screenhd: dc.l    0          ;идентификатор экрана
intbase:  dc.l    0          ;базовый адрес intuition

```

РАБОТА С INTUITION

end

Рассмотрим еще одну простую функцию Intuition — DisplayBeep (смещение -96 в intuition.library), параметром которой является идентификатор экрана. Эта функция на короткое время заполняет экран оранжевым (как правило) цветом, имитируя звуковой сигнал. Например:
DisplayBeep = -96

...

```
move.l    intbase(PC),A6          ;база intuition.library
move.l    screenhd(PC),A0         ;идентификатор экрана
jsr       DisplayBeep(A6)         ;"подсветить" экран
```

Если вместо идентификатора экрана задать ноль, "подсветится" вся видео- область.

Итак, теперь мы можем открывать экраны и манипулировать ими. Этого, однако, не достаточно для создания "полноценного" интерфейса. Конечно, Вы можете использовать собственные подпрограммы для вывода графики и текста на экран, но, согласитесь, гораздо проще и удобнее работать с библиотечными функциями, которые, к тому же, обеспечивают полную совместимость с операционной системой. Intuition содержит такие функции, однако многие из них работают с окнами, а не с экранами. В следующем пункте мы покажем, как открывать окна на произвольном экране.

7.2. Создание окон.

В пункте 6.3 мы рассматривали работу с окнами ввода/вывода, используя функции dos.library. На самом деле окно ввода/вывода — это простое окно Intuition, которое создается и управляется операционной системой, и поэтому не обладает всеми свойствами "нормального" окна. В частности, нельзя создать окно ввода/вывода на произвольном экране.

Изучив этот пункт, Вы сможете создавать окна на уровне Intuition. Для этого существует специальная функция — OpenWindow (смещение -204 в intuition.library), параметром которой является указатель на таблицу, описыва-

РАБОТА С INTUITION

ющую создаваемое окно.

Таблица параметров окна похожа на рассмотренную выше таблицу параметров экрана. Первые четыре слова определяют положение и размеры окна:

window_defs:

```
dc.w    10      ;X — координата
dc.w    20      ;Y — координата
dc.w    300     ;ширина окна (размер по X)
dc.w    150     ;высота окна (размер по Y)
```

Затем следуют два байта, определяющие цвета текста и фона:

```
dc.b    1      ;цвет текста (белый)
dc.b    3      ;цвет фона (красный)
```

Следующая запись определяет набор битов, влияющих на рассылку событий (сообщений) создаваемому окну (IDCMP — флаги). Каждый бит этого набора определяет, нужно ли посылать окну сообщение соответствующего типа:

Бит	Маска	Название	Тип события
0	\$000001	SIZEVERIFY	
1	\$000002	NEWSIZE	изменение размера
2	\$000004	REFRESHWINDOW	обновление содержимого окна
3	\$000008	MOUSEBUTTONS	нажатие кнопок мыши
4	\$000010	MOUSEMOVE	движение мыши
5	\$000020	GADGETDOWN	выбор специального gadget'a
6	\$000040	GADGETUP	то же
7	\$000080	REQSET	
8	\$000100	MENUPICK	выбор пункта меню
9	\$000200	CLOSEWINDOW	выбор gadget'a закрытия окна
10	\$000400	RAWKEY	нажатие клавиш клавиатуры
11	\$000800	REQVERIFY	
12	\$001000	REQCLEAR	
13	\$002000	MENUVERIFY	
14	\$004000	NEWPREFS	изменение установок (preferences)
15	\$008000	DISKINSERTED	установка диска
16	\$010000	DISKREMOVED	удаление диска
17	\$020000	WBFNCHMESSAGE	сообщение Workbench
18	\$040000	ACTIVIEWINDOW	активизация окна

РАБОТА С INTUITION

- 19 \$080000 INACTIVWINDOW деактивизация окна
 20 \$100000 DELTAMOVE относительное движение мыши

Например, если нам нужно создать окно, реагирующее только на выбор символа закрытия (close gadget), мы должны написать:

dc.l \$200 ;флаг IDCMP: CLOSEWINDOW'

Следующее двойное слово определяет тип окна:

Бит	Маска	Название	Назначение
0	\$0000001	WINDOWSIZING	размер окна может изменяться (присутствует gadget изменения размера)
1	\$0000002	WINDOWDRAG	положение окна может изменяться
2	\$0000004	WINDOWDEPTH	возможно закрывание окна другими окнами (присутствует gadget смены положения на сетке окон)
3	\$0000008	WINDOWCLOSE	присутствует gadget закрытия окна
4	\$0000010	SIZERIGHT	
5	\$0000020	SIZEBOTTOM	
6	\$0000040	SIMPLE_REFRESH	обычное обновление содержимого окна
7	\$0000080	SUPER_BITMAP	сохранение содержимого окна
8	\$0000100	BACKDROP	создать окно на заднем плане
9	\$0000200	REPORTMOUSE	окно может получать координаты курсора мыши
10	\$0000400	GIMMEZEROZERO	
11	\$0000800	BORDERLESS	бorders запрещен
12	\$0001000	ACTIVATE	создать и активизировать окно
13	\$0002000	WINDOWACTIVE	
14	\$0004000	INREQUEST	
15	\$0008000	MENUSTATE	
16	\$0010000	RMOTRAP	отмена реакции на правую кнопку мыши
17	\$0020000	NOCAREREFRESH	запрещено сообщение об обновлении
24	\$1000000	WINDOWREFRESH	
25	\$2000000	WBENCHWINDOW	

Обновление окна — это перерисовка его содержимого в случае необходимости (например, при изменении размеров окна). Если ни один из битов обновления в записи типа не установлен, включается режим “умной” перерисовки (Smart- Refresh-Mode). В этом случае Intuition берет на себя все действия по обновлению окна.

РАБОТА С INTUITION

Выберем, например, значение \$100F, которое задает активизированное окно с полным набором символов управления (gadgets):

dc.l \$100F ;флаги: ACTIVATE и все gadget'ы

Следующее слово должно содержать указатель на список специальных символов управления (gadgets), или ноль, если специальные символы не используются.

dc.l 0 ;не использовать специальные gadget'ы

Далее следует указатель на специальную графическую структуру, определяющую контрольные точки меню. Ноль, как обычно, задает стандартный вид контрольной точки:

dc.l 0 ;CheckBox: стандарт

Следующее слово должно содержать указатель на имя окна (ASCII):

dc.l windowname ;указатель на имя окна

Следующая запись определяет экран, на котором должно появиться окно, и содержит идентификатор экрана, возвращенный функцией OpenScreen. Будем использовать это поле для сохранения идентификатора:

screenhd: dc.l 0 ;идентификатор экрана

Следующее двойное слово определяет битовую карту, которая должна использоваться при работе с окном. Нулевое значение задает стандартную карту:

dc.l 0 ;использовать стандартный bitmap

Далее следуют четыре слова, определяющие максимальные и минимальные размеры окна:

dc.w 150 ;минимальная ширина

dc.w 50 ;минимальная высота

dc.w 320 ;максимальная ширина

dc.w 200 ;максимальная высота

Это означает, что окно не может быть увеличено до размеров, больших 320x200 (по одной из координат), и не может быть уменьшено до размеров, меньших 150x50.

И, наконец, последнее поле таблицы параметров задает тип экрана, на котором открывается окно (см. п. 7.1). Будем использовать полностью "настраиваемый" (custom)

РАБОТА С INTUITION

экран, для чего поместим сюда число 15:

dc.w 15 ;тип экрана: custom

Приведем полный список параметров окна:

window_defs:

```

dc.w 10 ;X — координата окна
dc.w 20 ;Y — координата окна
dc.w 300 ;ширина окна
dc.w 150 ;высота окна
dc.b 1 ;цвет текста
dc.b 3 ;цвет фона
dc.l $200 ;флаги IDCMP: реакция только на
;символ закрытия окна
dc.l $100F ;тип окна: используем все gadget'ы
dc.l 0 ;используем только стандартные
;gadget'ы
dc.l 0 ;используем стандартную контрольную
;точку (CheckMark) меню
dc.l windowname ;указатель на имя окна
screenhd: dc.l 0 ;идентификатор экрана
dc.l 0 ;используем стандартный bitmap
dc.w 150 ;минимальная ширина окна
dc.w 50 ;минимальная высота окна
dc.w 320 ;максимальная ширина окна
dc.w 200 ;максимальная высота окна
dc.w 15 ;тип экрана: custom windowname:
dc.b 'Our Window',0 ;ASCII-строка — имя окна
align
```

Для открытия и закрытия окна можно использовать следующие подпрограммы:

OpenWindow = -204

CloseWindow = -72

...

```

windopen: ;* открываем окно
move.l intbase(PC),A6 ;базовый адрес intuition
lea windowdefs(PC),A0 ;указатель на таблицу
;параметров окна
jsr OpenWindow(A6) ;открыть окно
move.l D0,windowhd ;сохраняем идентификатор окна
rts
```

РАБОТА С INTUITION

windowclose:

```
move.l intbase(PC),A6      ;базовый адрес intuition
move.l windowhd(PC),A0    ;идентификатор окна
jsr     CloseWindow(A6)   ;закрыть окно
rts
```

...

windowhd: dc.l 0 ;идентификатор окна

Модифицируйте приведенную в предыдущем пункте программу работы с экраном, добавив вызовы `windopen` и `windclose` после `scropen` и перед `scrclose` соответственно. После запуска полученной программы на экране появится окно, которое можно перемещать с помощью мыши (в пределах экрана).

Обратите внимание, что созданное окно имеет символ закрытия (`close gadget`) в левом верхнем углу. Обычно, при нажатии на этот символ окно закрывается, однако в нашем случае этого не произойдет. Реакция на любое событие (в частности, на нажатие символа закрытия) должна быть запрограммирована специальным образом. Об этом мы поговорим в следующих разделах.

7.3. Работа с реквестерами (requesters).

Если Вам приходилось работать с компьютером без жесткого диска (HD), то Вам наверняка знакомо сообщение "Please insert volume xxx in unit 0" (вставьте диск xxx в устройство 0). Эта надпись появляется в специальном окне операционной системы, которое также содержит поля ("кнопки") для выбора дальнейших действий (`Reset` и `Cancel`). Подобные окна называются реквестерами.

Для создания простого реквестера используется функция `AutoRequest` (смещение -348 в `intuition.library`). Эта функция рисует кнопки реквестера и обрабатывает события от мыши. Приведем список параметров этой функции:

- в `A0` — указатель на блок данных, описывающий окно (возвращается функцией `OpenWindow`)
- в `A1` — указатель на текст заголовка реквестера

РАБОТА С INTUITION

- в A2 — указатель на текст для левой кнопки реквестера
- в A3 — то же, только для правой кнопки
- в D0 — набор флагов, определяющих события для левой кнопки реквестера (формат IDCMP, см. п. 7.2)
- в D1 — то же, только для правой кнопки
- в D2 — ширина реквестера
- в D3 — высота реквестера

Эту функцию можно использовать, например, так:

AutoRequest = -348

...

request:

```

move.l    windowhd(PC),A0    ;идентификатор окна
lea       btext(PC),A1       ;текст реквестера
lea       ltext(PC),A2       ;текст для левой кнопки
lea       rtext(PC),A3       ;текст для правой кнопки
moveq.l   #0,D0              ;события для левой кнопки
                                ;реквестера: реакция — только
                                ;на нажатие левой кнопки мыши
moveq.l   #0,D1              ;события для правой кнопки
                                ;реквестера
move.l    #180,D2             ;ширина и
moveq.l   #80,D3             ;высота реквестера
move.l    intbase(PC),A6      ;база intuition.library
jsr       AutoRequest(A6)     ;создать реквестер
rts       ;возврат
    
```

Использование флагов в регистрах D0 и D1 дает некоторые интересные возможности. Например, если при появлении системного реквестера вставить диск в дисковод, реквестер исчезнет и система выполнит действия, эквивалентные выбору кнопки Retry. Это достигается путем задания флага DISKINSERTED в регистре D0.

Теперь рассмотрим формат текстовых сообщений, адреса которых указываются в регистрах A1, A2 и A3. Первые два байта каждого из этих сообщений определяют цвета текста и фона:

text:

```

dc.b      2                  ;цвет текста
dc.b      0                  ;текст фона
    
```


РАБОТА С INTUITION

Далее указывается режим вывода текста (например, режим 4 определяет инверсный вывод). Будем использовать обычный режим:

```
dc.b 0 ;обычный вывод текста
```

Следующие две записи определяют координаты текста в окне. Эти записи имеют формат Word (слово), а так как им предшествуют три байта, необходимо использовать директиву Align (для выравнивания по четности):

```
align
dc.w 10 ;X — координата текста
dc.w 5 ;Y — координата текста
```

Далее следует указатель на таблицу символов (шрифт). Ноль, как обычно, задает стандартный шрифт:

```
dc.l 0 ;используем стандартный шрифт
```

Следующая запись должна содержать указатель на выводимую строку, заданную в ASCII-формате:

```
dc.l text ;указатель на текст реквестера
```

Завершает таблицу указатель на дополнительную строку текста. Если дополнительный текст не нужен, эта запись должна содержать ноль:

```
dc.l 0 ;не использовать дополнительный текст
```

Приведем пример текстовых структур, которые могут использоваться в функции AutoRequest:

```
btext:
    dc.b 0,1 ;цвета текста и фона
    dc.b 0 ;режим вывода
    align
    dc.w 10,10 ;положение текста
    dc.l 0 ;шрифт — стандартный
    dc.l bodytxt ;указатель на ASCII-строку
    dc.l 0
bodytxt:
    dc.b 'Requester text',0
    align
ltext:
    dc.b 0,1 ;цвета текста и фона
    dc.b 0 ;режим вывода
    align
    dc.w 5,3 ;положение текста
```

РАБОТА С INTUITION

```

dc.l      0      ;шрифт
dc.l      lefttext ;указатель на текст
dc.l      0
lefttext: dc.b    'left',0
          align
rtext:    .      ;текст для правой кнопки реквестера
          dc.b    0,1 ;цвета текста и фона
          dc.b    0      ;режим вывода текста
          align
          dc.w    5,3    ;положение текста
          dc.l    0      ;шрифт
          dc.l    righttext ;указатель на текст
          dc.l    0
righttext: dc.b    'right',0
          align

```

После вызова `AutoRequest` регистр `D0` будет содержать информацию о выборе пользователя: правой кнопке реквестера соответствует ноль, левой — единица.

7.4. Обработка событий.

Представьте, что Вы создали окно с символом закрытия и хотите обрабатывать событие, соответствующее “нажатию” этого символа. Для этого в `Intuition` используется механизм сообщений: если пользователь “нажимает” (с помощью мыши) на какой-либо символ (`gadget`) окна, то программе, создавшей это окно, посылается соответствующий сигнал. Этот сигнал называется сообщением (`message`).

В предыдущем пункте мы рассматривали таблицу параметров окна, одним из полей которой является так называемое `IDCMP`-слово. Это слово определяет события, которые могут вызывать посылку сообщений данному окну. Например, если в `IDCMP`-слове установлен бит `WINDOW-CLOSE` (см. п. 7.2), то окно сможет “узнать” о нажатии символа закрытия.

Чтобы получить сообщение, программа должна вызвать функцию `GetMsg` (смещение -372 в `exec.library`). Параметром этой функции является адрес специальной структуры, которая после вызова `GetMsg` будет содержать ин-

РАБОТА С INTUITION

формацию о событии, вызвавшем посылку сообщения. Эта структура называется пользовательским портом (user port).

Адрес пользовательского порта можно узнать, используя идентификатор окна. На самом деле идентификатор окна — это указатель на специальную структуру, которая содержит всю информацию об окне, и в частности, адрес пользовательского порта (смещение 86 относительно начала структуры). Извлечь этот адрес можно, например, так:

```
move.l    windowhd(PC),A0    ;идентификатор окна — адрес
                                ;блока данных окна
move.l    86(A0),A0           ;извлекаем адрес
                                ;пользовательского порта
```

После этого можно вызывать функцию GetMsg.
GetMsg = -372

```
...
move.l    ExecBase,A6         ;база exec.library
jsr       GetMsg(A6)           ;получаем сообщение
```

Функция GetMsg возвращает в D0 указатель на специальную структуру — IMS (Intuition Message Structure), содержащую информацию о полученном сообщении, либо ноль, если сообщения не было.

Двойное слово, расположенное с 20-го байта структуры IMS, содержит информацию о событиях, вызвавших посылку сообщения. Биты этого слова имеют тот же смысл, что и IDCMP-флаги, рассмотренные ранее. Так, для проверки нажатия символа закрытия, добавьте в программу 7.1 (после метки loop) следующий фрагмент:

```
move.l    D0,A0               ;указатель на IMS — в A0
move.l    20(A0),D6           ;информация о событии
tst.l     D0                  ;пришло сообщение?
bnc.s     end                 ;да, выход
```

Вы можете использовать регистр D6 для получения информации о событии. В нашем примере D6 будет содержать число \$00000200, что сигнализирует о нажатии символа закрытия. Так как при создании окна мы указали реакцию только на это событие, D6 не может содержать дру-

РАБОТА С INTUITION

гих ненулевых значений.

Теперь рассмотрим другой случай. Измените значение IDCMP-слова с \$200 на \$10200 и запустите программу. Удалите диск из дисководов — и программа завершится. Это происходит потому, что помимо флага реакции на нажатие символа закрытия (\$200) мы указали флаг DISKREMOVED (\$10000), который соответствует событию удаления диска.

Если в D6 возвращается ненулевое значение, то возможны следующие варианты: D6 содержит \$200 — тогда произошло событие, соответствующее нажатию символа закрытия. Если же D6 содержит \$10000, то удален диск. Так как мы указали только эти два флага в таблице параметров, функция GetMsg не может возвращать других ненулевых значений.

7.5. Работа с меню.

В этом пункте мы рассмотрим набор средств Intuition, позволяющих работать с одним из важнейших элементов графического интерфейса — меню. Согласитесь, что трудно представить системную программу на Amiga, не использующую меню.

Существует множество способов создания и использования меню. Например, можно создавать меню сложной структуры с наличием подменю; выбирать тип пунктов меню (текст или графика), итд. Intuition имеет очень гибкие средства для работы с меню, поэтому для многих функций требуются громоздкие таблицы параметров.

Для начала рассмотрим простейший способ создания меню. В состав intuition.library входит функция SetMenuStrip (смещение -264), которая вызывается с двумя параметрами:

- в A0 — указатель на структуру окна (идентификатор окна), для которого нужно создать меню
- в A1 — указатель на блок данных, определяющий структуру меню

Каждое окно может иметь собственное меню верхнего уровня (горизонтальное), которое становится доступ-

РАБОТА С INTUITION

ным одновременно с активизацией окна.

Функцию SetMenuStrip можно использовать, например, так:

SetMenuStrip = -264

```

...
setmenu:                                ;* инициализация меню
      move.l    intbase(PC),A6           ;база intuition.library
      move.l    windowhd(PC),A0         ;идентификатор окна
      lea       menu(PC),A1             ;указатель на блок данных меню
      jsr       SetMenuStrip(A6)         ;вызываем SetMenuStrip
      rts                                     ;все
  
```

Для уничтожения меню используется функция

ClearMenuStrip:

ClearMenuStrip = -54

```

...
clearmenu:                              ;* уничтожение меню
      move.l    intbase(PC),A6           ;база intuition.library
      move.l    windowhd(PC),A0         ;идентификатор окна
      jsr       ClearMenuStrip(A6);уничтожаем меню
      rts
  
```

Для каждого пункта меню верхнего уровня должна быть определена специальная структура — блок данных меню. Рассмотрим формат этого блока:

Первое слово (формат Long) содержит указатель на блок данных следующего пункта меню, либо ноль, если текущий пункт является последним.

```

menu:
      dc.l      menu1                   ;указатель на следующее меню
  
```

Далее следуют два слова, определяющие координаты

заголовка меню:

```

      dc.w      20                      ;X — координата
      dc.w      0                      ;Y — координата
  
```

Следующие два слова содержат ширину и высоту заголовка меню (в пикселах):

```

      dc.w      50                      ;ширина заголовка
      dc.w      10                      ;высота заголовка
  
```

Следующее слово содержит специальный флаг, определяющий, доступно данное меню или нет (заголовки не-

РАБОТА С INTUITION

доступных меню обычно отображаются "прерывистым" шрифтом). Если этот флаг (нулевой бит) установлен, то меню доступно.

dc.w 1 ;меню доступно

Следующее двойное слово должно содержать указатель на текст заголовка меню (убедитесь, что ширина этого текста не превосходит значения ширины заголовка, указанного ранее):

dc.l menuitem1 ;указатель на текст заголовка меню

Далее следует указатель на блок данных первого пункта текущего меню. Для каждого пункта меню должен быть определен блок данных, структура которого будет рассмотрена ниже.

dc.l menuitem01 ;указатель на первый пункт меню,
;который, в свою очередь, тоже
;может содержать меню

И, наконец, последние четыре записи зарезервированы для внутреннего использования:

dc.w 0,0,0,0 ;зарезервировано

Блок данных, структуру которого мы рассмотрели, используется только для пунктов меню верхнего уровня иерархии. Блоки данных "выпадающих" меню имеют несколько другой формат:

Первое слово должно содержать указатель на следующий пункт определяемого "выпадающего" (вертикального) меню, либо ноль, если текущий пункт является последним:

menuitem01:

dc.l menuitem02 ;указатель на следующий пункт

Следующие четыре слова определяют координаты и размеры заголовка меню (координаты указываются относительно левого верхнего угла "выпадающего" меню):

dc.w 0 ;X — координата

dc.w 0 ;Y — координата

dc.w 90 ;ширина — 90 пикселей

dc.w 10 ;высота — 10 пикселей

Следующее слово содержит набор флагов, задающих

РАБОТА С INTUITION

режим работы с текущим пунктом меню:

Бит	Маска	Имя	Назначение (если бит установлен)
0	\$0001	CHECKIT	добавление символа проверки при выборе пункта
1	\$0002	ITEMTEXT	пункт меню содержит обычный текст
2	\$0004	COMSEQ	допускается выбор по комбинации "горячих" клавиш
3	\$0008	MENUTOGGLE	пункт меню является "переключателем"
4	\$0010	ITEMENABLED	пункт меню доступен
6	\$0040	HIGHCOMP	инвертирование заголовка при выборе пункта
7	\$0080	HIGHBOX	выделение заголовка с помощью прямоугольника
8	\$0100	CHECKED	наличие символа проверки

Приведем более подробное описание рассмотренных флагов:

CHECKIT	если этот флаг установлен, при выборе пункта меню перед заголовком будет добавляться символ проверки. Текст заголовка должен начинаться с двух пробелов.
ITEMTEXT	определяет тип заголовка меню. Если этот флаг установлен, текущий пункт меню содержит обычный текст, иначе допускается использование графики.
COMSEQ	если этот флаг установлен, текущий пункт меню может быть активизирован нажатием комбинации клавиш <Left Amiga> + <любая определенная пользователем клавиша>. Заголовок меню должен содержать необходимое количество пробелов для отображения символов "горячей" комбинации.
MENUTOGGLE	если этот бит установлен и разрешено отображение символа проверки (установлен флаг CHECKIT), то текущий пункт меню будет работать как "переключатель":

РАБОТА С INTUITION

каждый последующий выбор этого пункта будет изменять его состояние на противоположное (возможные состояния: "включен" и "выключен").

- | | |
|-------------|--|
| ITEMENABLED | если этот флаг очищен, текущий пункт меню становится недоступным. |
| HIGHCOMP | установка этого бита включает режим инвертирования заголовка текущего пункта при его выборе. |
| HIGHBOX | включает режим выделения заголовка с помощью прямоугольника. |

Последние два бита определяют режим выделения пункта меню при попадании указателя мыши в поле его заголовка. Помимо описанных, возможны следующие комбинации:

- | | |
|-----------|---|
| HIGHIMAGE | если оба флага (HIGHCOMP и HIGHBOX) очищены, выбор текущего пункта меню будет сопровождаться выводом произвольного графического изображения. |
| HIGHNONE | если флаги HIGHCOMP и HIGHBOX установлены, определяемый пункт меню никак выделяться не будет. |
| CHECKED | этот флаг может быть установлен как программой пользователя, так и системой Intuition. Если текущий пункт меню является переключателем, то флаг CHECKED определяет его состояние (1 - включен, 0 - выключен). |

Выберем, например, следующую комбинацию флагов:

dc.w %10011111

Вернемся к описанию структуры блока данных "выпадающих" меню. Следующее двойное слово определяет связи текущего пункта меню с другими пунктами:

dc.l 0 ;нет связей

Далее следует указатель на текст заголовка определяемого пункта меню, либо на графическую структуру, ес-

РАБОТА С INTUITION

ли флаг ITEMTEXT очищен. Например:

```
dc.l menu01text ;указатель на текст заголовка
```

Если говорить более точно, здесь указывается адрес не ASCII-строки текста, а специальной структуры, определяющей текст заголовка. Формат этой структуры мы описывали в п. 7.3.

Следующее двойное слово имеет смысл только в том случае, если установлен флаг HIGHIMAGE, и содержит указатель на графическую структуру, которая должна использоваться для выделения текущего пункта меню при его выборе. Если флаг HIGHIMAGE очищен, эта запись игнорируется, и, следовательно, может содержать любое значение:

```
dc.l 0 ;не использовать графику
```

Следующий байт определяет клавишу, которая должна использоваться в "горячей" комбинации (вместе с <Left Amiga>) для выбора текущего пункта меню, и имеет смысл только в том случае, если флаг COMMSEQ установлен:

```
dc.b 'A' ;выбираем комбинацию <Amiga> + 'A'
```

Следующее двойное слово определяет подменю, которое должно активизироваться при выборе текущего пункта меню. Это слово должно содержать указатель на блок данных подменю (который имеет ту же структуру, что и описываемый в данный момент), либо ноль, если подменю не требуется. Заметим, что допускается только один уровень вложенности меню, так что для подменю этот указатель игнорируется.

```
align
```

```
dc.l 0 ;не показывать подменю
```

Здесь мы используем директиву align, так как предыдущая запись имеет размер 1 байт.

И завершает описываемый блок данных двойное слово, которое используется Intuition при выборе сразу нескольких пунктов меню:

```
dc.l 0 ;заполняется системой Intuition
```

РАБОТА С INTUITION

В качестве примера рассмотрим полный набор данных для определения двух пунктов меню верхнего уровня, каждый из которых состоит из двух подпунктов. Второй подпункт правого меню, в свою очередь, содержит подменю, состоящее из двух пунктов. Мы рекомендуем Вам ввести приводимый блок данных в компьютер и поэкспериментировать с функцией SetMenuStrip.

;** Complete menu structure for example menu **

```
menu:
    dc.l    menu1          ;следующий пункт меню верхнего
                           ;уровня
    dc.w    10,30          ;координаты заголовка
    dc.w    50,10          ;размеры заголовка
    dc.w    1              ;меню доступно
    dc.l    menuname       ;текст заголовка меню
    dc.l    menuitem01     ;подпункты меню
    dc.w    0,0,0,0        ;зарезервировано menuname:
    dc.b    'Menu1',0      ;заголовок первого меню
    align

menu1:
    dc.l    0              ;последний пункт
    dc.w    80,0           ;координаты заголовка
    dc.w    50,10         ;размеры заголовка
    dc.w    1              ;меню доступно
    dc.l    menuname1     ;текст заголовка меню
    dc.l    menuitem11     ;подпункты меню
    dc.w    0,0,0,0        ;зарезервировано menuname1:
    dc.b    'Menu2',0     ;заголовок второго меню
    align

menuname1:
    dc.b    'Menu2',0     ;заголовок второго меню
    align

menuitem01:
    dc.l    menuitem02    ;следующий подпункт
    dc.w    0,0           ;координаты заголовка
    dc.w    130,12        ;размеры
    dc.w    $9F           ;флаги
    dc.l    0              ;нет связей
    dc.l    text01        ;текст заголовка
    dc.l    0              ;координаты заголовка
    dc.b    '1'           ;"горячая" клавиша
```

РАБОТА С INTUITION

```

align
dc.l      0           ;нет подменю
dc.l      0
text01:
dc.b      0,1         ;цвета текста и фона
dc.b      0           ;режим
align
dc.w      5,3         ;координаты
dc.l      0           ;стандартный шрифт
dc.l      text01txt   ;ASCII-строка заголовка
dc.l      0
text01txt:
dc.b      'Point0.1',0
align
menuitem02:          ;второй подпункт первого меню
dc.l      0
dc.w      0,10
dc.w      130,12
dc.l      0
dc.l      text02
dc.l      0
dc.b      '2'
align
dc.l      0
dc.l      0
text02:
dc.b      0,1
dc.b      0
align
dc.w      5,3
dc.l      0
dc.l      text02txt
dc.l      0
text02txt:
dc.b      'Point0.2',0
align
menuitem11:          ;первый подпункт второго меню
dc.l      menuitem12 ;указатель на второй подпункт
dc.w      0,0
dc.w      90,12

```

AMIGA: ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ.

РАБОТА С INTUITION

```
dc.w    $52
dc.l    0
dc.l    text11
dc.l    0
dc.b    0
align
dc.l    0
dc.l    0
text11:
dc.b    0,1
dc.b    0
align
dc.w    5,3
dc.l    0
dc.l    text11txt
dc.l    0
text11txt:
dc.b    'Point1.1',0
align
menuitem12:                ;второй подпункт второго меню
dc.l    0                    ;последний подпункт
dc.w    0,10
dc.w    90,12
dc.w    $92
dc.l    0
dc.l    text12
dc.l    0
dc.b    0
align
dc.l    submenu0            ;указатель на подменю
dc.l    0
text12:
dc.b    0,1
dc.b    0
align
dc.w    5,3
dc.l    0
dc.l    text12txt
dc.l    0
text12txt:
```

РАБОТА С INTUITION

```

dc.b      'Point1.2',0
align
submenu0:                                :первый пункт подменю
dc.l      submenu1                      :указатель на второй пункт
dc.w      80,5
dc.w      90,12
dc.w      $52
dc.l      0
dc.l      texts0
dc.l      0
dc.b      0
align
dc.l      0
dc.l      0
texts0:
dc.b      0,1
dc.b      0
align
dc.w      5,3
dc.l      0
dc.l      texts0txt
dc.l      0
texts0txt:
dc.b      'SPoint1',0
align
submenu1:                                :второй пункт подменю
dc.l      0
dc.w      80,15
dc.w      90,12
dc.w      $52
dc.l      0
dc.l      texts1
dc.l      0
dc.b      0
align
dc.l      0
dc.w      0
texts1:
dc.b      0,1
dc.b      0

```

РАБОТА С INTUITION

```
align
dc.w      5,3
dc.l      0
dc.l      texts1txt
dc.l      0
texts1txt:
dc.b      'SPoint2',0
align
```

Определенные таким образом меню обладают следующими свойствами:

Меню 1: Первый пункт этого меню, 'Point0.1', является переключателем и может быть активизирован нажатием комбинации клавиш <Left Amiga> + '1'. При выборе этого пункта с помощью мыши его заголовок выделяется прямоугольником.

Второй пункт, 'Point0.2', может быть активизирован нажатием "горячей" комбинации <Left Amiga> + '2', а при выборе с помощью мыши его заголовок выделяется инвертированием. Этот пункт содержит символ проверки (check symbol), однако не является переключателем — его состояние нельзя изменить.

Меню 2: Пункты этого меню не могут быть выбраны с помощью клавиатуры. Для выделения заголовка первого пункта используется инвертирование, второй пункт выделяется прямоугольником. При выборе второго пункта ('Point1.2') появляется подменю, содержащее два пункта.

До сих пор мы рассматривали только действия по созданию меню нужной структуры. Однако возникает вопрос: как программа пользователя "узнает" о выборе того или иного пункта меню?

Один из способов проверки состояния меню основывается на тестировании бита CHECKED в слове, определяющем флаги нужного пункта меню: как только пользователь выбирает этот пункт с помощью мыши, бит CHECKED устанавливается.

Однако этот способ имеет ряд недостатков. Во-первых, он работает только в том случае, если установлен флаг

РАБОТА С INTUITION

разрешения проверки. Во-вторых, он требует большого числа проверок, а именно, бит CHECKED каждого пункта меню и подменю должен быть протестирован отдельно. Все это приводит к очень громоздкому коду, поэтому мы рассмотрим другой способ (который и используется на практике).

Вспомним формат IDCMP-слова, рассмотренный нами в п. 7.2. Это слово, как Вы уже знаете, содержит набор флагов, определяющих, какие события может принимать данное окно. Нас будет интересовать восьмой бит этого слова (MENUPICK): если этот бит установлен, то при выборе какого-либо пункта меню функция GetMsg возвратит информацию об этом пункте в специальном поле IMS-структуры (см. п. 7.4). Поместите следующий фрагмент кода в программу из п. 7.1 (после метки loop):

```
move.l   ExecBase(PC),A6
move.l   windowhd(PC),A0 ;идентификатор окна
move.l   86(A0),A0        ;адрес пользовательского порта
jsr      GetMsg(A6)        ;получаем сообщение
tst.l    D0
beq.s    loop              ;сообщения не было
move.l   D0,A0             ;указатель на IMS-структуру
move.l   20(A0),D6         ;тип события — в D6
```

После этого мы можем использовать команду CMP или BTST для проверки содержимого регистра D6. Например:

```
cmp      #$200,D6          ;WINDOWCLOSE?
beq.s    ende              ;да, выход
```

Здесь производится проверка нажатия символа закрытия окна и выход, если соответствующее событие имело место.

Если пользователь выбрал какой-либо пункт меню, в регистре D6 возвратится значение \$100 (маска, соответствующая установленному биту MENUPICK). Нам остается только определить, какой пункт был выбран.

Информация о выбранном пункте содержится в слове, которое следует за полем типа события в структуре IMS (смещение 24):

```
move     24(A0),D7
```

РАБОТА С INTUITION

Если никакой пункт никакого меню не был выбран, но была нажата и отпущена правая кнопка мыши, в регистр D7 запишется \$FFFF. Иначе D7 будет содержать информацию о выбранном пункте, а именно, о номер меню, в которое входит выбранный пункт о номер подменю, в которое входит выбранный пункт о номер выбранного пункта

Значение, возвращаемое в регистре D7, имеет следующий формат:

- биты 0-4 номер меню верхнего уровня
- биты 5-10 номер пункта в меню верхнего уровня
- биты 11-15 номер пункта в подменю

Заметим, что нумерация начинается с нуля, так что при выборе первого пункта первого меню в D7 возвратится ноль.

Для выделения номеров пункта, меню и подменю будем использовать следующий код:

```
move    D7,D5
and     #$1F,D5      ;в D5 — номер меню верхнего
                     ;уровня

lsr     #5,D7
move    D7,D6
and     #$3F,D6      ;в D6 — номер пункта в меню
                     ;верхнего уровня

lsr     #6,D7      ;в D7 — номер пункта в подменю
cmp     #$3F,D6    ;пункт меню выбран?
beq.s   loop       ;нет, цикл
```

Представьте теперь, что Вы хотите создать четыре меню верхнего уровня, каждое из которых должно содержать по 10 пунктов. Для этого Вам потребуются 44 таблицы, определяющие блоки данных для каждого пункта. Согласитесь, что задавать такое количество таблиц вручную занятие весьма нудное. Поэтому мы приведем небольшую, но полезную программу, которая создает таблицы нужной структуры на основании простого списка пунктов меню.

Наша программа может строить таблицы только для простых меню (а именно, не содержащих подменю и "горячих" клавиш). Однако при желании Вы можете самостоя-

РАБОТА С INTUITION

тельно добавить недостающие функции.

Исходный список пунктов меню для нашей программы имеет следующий формат:

```
dc.l  MenuTitle1           ;текст заголовка первого меню
dc.l  Point11,Point12,Point13,...,0 ;пункты первого меню
dc.l  MenuTitle2           ;текст заголовка второго меню
dc.l  Point21,Point22,Point23,...,0 ;пункты второго меню
...
dc.l  0
```

Программа может обрабатывать не более четырех меню верхнего уровня (в большинстве случаев этого вполне достаточно). Признаком окончания списка меню служит нулевой указатель.

Приведем текст этой программы:

```
setmenu:                ;* создает и инициализирует блоки
                        ;* данных меню
        lea     mentab(PC),A0 ;указатель на исходный список
                        ;пунктов меню
        lea     menu(PC),A1  ;указатель на блок данных меню
        moveq   #10,D1       ;горизонтальная позиция меню
menuloop:
        moveq   #0,D2        ;вертикальная позиция меню
        move.l  A1,A2
        tst.l   (A0)         ;конец списка меню?
        beq     setmenu1     ;да, выход
        clr.l   (A1)+        ;нет других пунктов
        move    D1,(A1)+     ;X — координата
        add.l   #70,D1       ;смещение по X
        move.l  #50,(A1)+    ;Y — координата и ширина
        move.l  #$A0001,(A1)+ ;высота и флаги
        move.l  (A0)+,(A1)+  ;заголовок меню
        lea     12(A1),A3    ;A3 = A1 + 12
        move.l  A3,(A1)+    ;указатель на пункт меню
        clr.l   (A1)+       ;зарезервированные слова
        clr.l   (A1)+
itemloop:
        tst.l   (A0)         ;конец списка пунктов?
        beq     menuend     ;да, переходим к следующему
                        ;меню
```

РАБОТА С INTUITION

```

lea      54(A1),A3      ;A3 = A1 + 54
move.l   A3,(A1)+       ;указатель на следующий пункт
move.l   D2,(A1)+       ;X и Y координаты
add      #10,D2         ;смещение по Y
move.l   #55A000A,(A1)+;ширина/высота
move     #552,(A1)+     ;флаги
clr.l    (A1)+          ;нет связей
lea      16(A1),A3      ;A3 = A1 + 16
move.l   A3,(A1)+       ;указатель на текстовую
                        ;структуру заголовка

clr.l    (A1)+
clr.l    (A1)+          ;нет "горячей" комбинации и
clr.l    (A1)+          ;подменю
move     #1,(A1)+       ;определяем структуру
                        ;заголовка: цвета текста и фона

clr      (A1)+          ;режим
move.l   #550003,(A1)+  ;X и Y координаты
clr.l    (A1)+          ;стандартный шрифт
move.l   (A0)+,(A1)+    ;текст заголовка
clr.l    (A1)+          ;конец текста
bra.s    itemloop       ;цикл по пунктам меню

menuend:
clr.l    -54(A1)        ;очищаем указатель на
                        ;следующий пункт
tst.l    (A0)+          ;увеличиваем указатель списка
tst.l    (A0)           ;конец списка?
beq.s    setmenu1       ;да, выход
move.l   A1,(A2)        ;иначе сохраняем указатель на
                        ;следующее меню верхнего уровня
bra menuloop            ;цикл

setmenu1:                ;* создать меню
move.l   intbase(PC),A6 ;база intuition.library
move.l   windowhd(PC),A0;идентификатор окна
lea      menu(PC),A1     ;указатель на блок данных меню
jsr      SetMenuStrip(A6);создаем меню
rts      ;все!

```

В конце программы, как всегда, приводится блок переменных и констант:

```

mentab:                ;тестовый список меню
dc.l     menu1         ;заголовок первого меню

```

РАБОТА С INTUITION

```

dc.l mp11,mp12,mp13      ;пункты первого меню
dc.l 0                    ;конец списка пунктов
dc.l menu2                ;заголовок второго меню
dc.l mp21,mp22,mp23      ;пункты второго меню
dc.l 0                    ;конец списка пунктов
dc.l 0                    ;конец списка меню
; ** Menu Text **
menu1: dc.b 'Menu1',0      ;заголовок первого меню
mp11:  dc.b 'Point11',0    ;пункты первого меню
mp12:  dc.b 'Point12',0
mp13:  dc.b 'Point13',0
menu2:  dc.b 'Menu2',0     ;заголовок второго меню
mp21:  dc.b 'Point21',0    ;пункты второго меню
mp22:  dc.b 'Point22',0
mp23:  dc.b 'Point23',0
align
; ** Storage space for menu structure **
menu:  blk.w 500           ;место для блоков данных

```

Попробуйте изменить эту программу так, чтобы она допускала использование "горячих" клавиш. Для этого, кстати вовсе необязательно переписывать всю программу: Вы можете добавить недостающие поля структур сразу после их построения. Адреса этих полей можно легко вычислить, используя формат блоков данных меню, который мы подробно рассматривали выше. Например, адрес байта "горячей" клавиши для второго пункта первого меню можно вычислить по формуле:

$$\text{Address} = \text{Start_address} + \text{Menu} * 30 + (\text{Entry} - 1) * 54 + 26$$

, где Start_address — начальный адрес блока данных,

Menu — номер меню,

Entry — номер пункта.

Подставляя значения Menu = 1 и Entry = 2, получим:

$$\text{Address} = \text{menu} + 30 + 54 + 26 = \text{menu} + 110$$

Кстати, не забудьте установить флаг реакции на "горячие" клавиши, иначе проведенные выше изменения байта кода клавиши не будут иметь смысла.

Итак, мы рассмотрели все функции, необходимые

РАБОТА С INTUITION

для работы с экранами, окнами и меню, кроме функций ввода/вывода. Следующие два пункта мы посвятим описанию средств Intuition, предназначенных для работы с текстом и графикой.

7.6. Вывод текста.

Для вывода текста в произвольное окно Intuition используется функция `PrintText` (смещение -216 в `intuition.library`), которая вызывается с четырьмя параметрами:

- в `A0` — указатель на порт окна (`RastPort`)
- в `A1` — указатель на блок данных, определяющий выводимый текст. Структуру этого блока мы уже рассматривали в п. 7.3.
- в `D0` — `X` - координата текста
- в `D1` — `Y` - координата текста

Порт окна (`RastPort`) — это специальная структура, содержащая различную служебную информацию об окне. Почти все функции Intuition, работающие с графикой и текстом, требуют адрес порта окна в качестве параметра.

Этот адрес содержится в блоке данных, на который указывает идентификатор окна (смещение 50), и может быть считан следующими командами:

```
move.l windowhd(PC),A0 ;идентификатор окна
move.l 50(A0),A0        ;адрес порта RastPort
```

Остальные параметры функции `PrintText` нам уже знакомы, так что теперь мы можем написать подпрограмму, выводящую текст в окно Intuition. Пусть перед вызовом этой подпрограммы регистры `D0` и `D1` содержат `X` и `Y` координаты текста, а регистр `A1` содержит адрес блока данных текста:

`PrintText = -216`

print:

```
move.l intbase(PC),A6 ;база intuition.library
move.l windowhd(PC),A0 ;идентификатор окна
move.l 50(A0),A0       ;адрес порта RastPort
```

РАБОТА С INTUITION

```
jsr   PrintText(A6)      ;выводим текст
rts
```

7.7. Работа с графикой.

В этом пункте мы рассмотрим функцию библиотеки intuition.library, которая позволяет выводить произвольные графические изображения в окна Intuition. Под графическим изображением мы будем понимать прямоугольную область, содержащую графическую информацию в обычном побитовом формате (каждому пикселу области соответствует один бит).

Размеры изображений могут варьироваться, однако для задания больших изображений требуется слишком громоздкий текст (на практике графические изображения редко задаются с помощью директив типа dc.l, вместо этого используется директива ассемблера incbin, которая позволяет включать в код программы двоичные файлы). По этой причине мы будем рассматривать изображения небольших размеров (32x16 точек — около 3x1 см.).

Для вывода изображения в произвольное окно Intuition может использоваться функция DrawImage (смещение -114 в intuition.library). Параметры у этой функции следующие:

- в A0 — адрес порта окна (RastPort), см. предыдущий пункт
- в A1 — адрес блока данных, определяющего изображение
- в D0 — X - координата изображения (левого верхнего угла)
- в D1 — Y - координата изображения

Рассмотрим простую подпрограмму вывода изображения в окно Intuition (будем предполагать, что регистры D0 и D1 содержат координаты изображения, а регистр A1 содержит адрес графической структуры):

DrawImage = -114

```
...
draw:                                     ;* нарисовать изображение (Image)
      move.l   intbase(PC),A6             ;база intuition.library
      move.l   windowhd(PC),A0           ;идентификатор окна
      move.l   50(A0),A0                 ;адрес порта RastPort
```

РАБОТА С INTUITION

jsr DrawImage(A6) ;рисуем изображение
rts

Теперь рассмотрим структуру блока данных, описывающего изображение.

Первые два слова определяют смещения (по X и по Y) изображения относительно заданных в D0 и D1 координат:

image:

dc.w 0,0 ;смещение изображения

Следующие два слова определяют ширину и высоту изображения в пикселах:

dc.w 32,13 ;ширина и высота изображения

Следующее слово задает число битпланов для вывода изображения. От числа битпланов зависит количество цветов, которые могут использоваться в изображении: чем больше битпланов, тем больше цветов доступно. Однако есть и обратный эффект: большое количество битпланов требует большего объема памяти для хранения изображений. Для простоты будем использовать один битплан:

dc.w 1 ;один битплан, $2^1 = 2$ цвета

Это вовсе не означает, что заданное таким образом изображение может выводиться только в те окна и экраны, "глубина" которых равна 1. Количество битпланов изображения может быть меньше, чем "глубина" окна.

Далее следует указатель на данные, определяющие изображение:

dc.l imgdata ;"биты" изображения

Следующий байт (PlanePick byte) определяет набор битпланов, в которые должно выводиться изображение. При этом каждый бит байта PlanePick соответствует отдельному битплану: бит 0 задает нулевой битплан, бит 1 — первый, итд. Фактически, данный байт определяет цвет выводимого изображения.

dc.b 2 ;рисуем в первый битплан (красный
;цвет, см. примеры из предыдущих
;пунктов)

Биты следующего байта (PlaneOnOff byte) опреде-

РАБОТА С INTUITION

ляют битпланы, которые должны заполняться единицами или нулями независимо от того, какое значение имеют соответствующие биты структуры `imgdata`. Этот байт может использоваться для очистки битпланов, которые не заданы в байте `PlanePick`, и, следовательно, не участвуют в выводе изображения. В нашем примере существует один такой битплан — нулевой (так как “глубина” окна равна двум). Установим нулевой бит байта `PlaneOff`, в результате чего нулевой битплан заполнится единицами:

```
dc.b 1 ;цвет фона: белый
```

И, наконец, последняя запись рассматриваемого блока данных содержит указатель на другое изображение:

```
dc.l 0 ;больше нет изображений
```

Итак, мы получили следующую структуру:

`image:`

```
dc.w 0,0 ;смещения по X и Y
dc.w 32,13 ;размеры изображения
dc.w 1 ;1 битплан
dc.l imgdata ;указатель на данные изображения
dc.b 2 ;рисуем в первом битплане
dc.b 1 ;заполняем нулевой битплан единицами,
dc.l 0 ;больше нет изображений
```

Теперь рассмотрим формат блока данных изображения (`imgdata`). Данные изображения содержат последовательность байтов (слов, двойных слов), отдельные биты которых соответствуют пикселям изображения (если бит установлен, соответствующий пиксел “подсвечивается”). Изображение задается слева направо, строка за строкой (сверху вниз).

В нашем примере изображение задается достаточно легко, так как каждой его строке будет соответствовать ровно четыре байта (32 бита — размер изображения по X). В качестве примера такого изображения рассмотрим переключатель, находящийся в состоянии “OFF” (выключено). В пункте, посвященном `gadget`’ам, мы покажем, как перевести этот переключатель в состояние “ON”. А пока определим блок `imgdata` следующим образом:

РАБОТА С INTUITION

```
imgdata:                                ;изображение переключателя "OFF"
dc.l      %00000000000000000000000000000000
dc.l      %000000000000000000000000111000000000
dc.l      %0001110111011100000111110000000000
dc.l      %0001010100010000000011111000000000
dc.l      %0001010110011000000111100000000000
dc.l      %0001110100010000000111000000000000
dc.l      %0000000000000000000111000000000000
dc.l      %0000000000000000000111000000000000
dc.l      %0000000000001111111111100000000000
dc.l      %0000000000111111111111000000000000
dc.l      %0000000000011111111111100000000000
dc.l      %0000000000011000000110000000000000
dc.l      %00000000000000000000000000000000
```

Для вывода этого изображения вставьте в программу из предыдущего пункта следующие строки (перед меткой loop):

```
lea      image(PC),A1          ;указатель на структуру image
moveq    #30,D0                ;X — координата в окне
moveq    #50,D1                ;Y — координата в окне
bsr.s    draw                  ;рисуем изображение
```

7.8. Работа с бордюрами.

Бордюр — это набор связанных друг с другом отрезков. Эти отрезки могут располагаться под произвольными углами и иметь произвольную длину. В системе Intuition бордюры используются для выделения окон и экранов, а также для обозначения поля ввода для строковых gadget'ов.

Нарисовать бордюр очень несложно: достаточно воспользоваться функцией DrawBorder (смещение -108 в intuition.library), которая вызывается с тремя параметрами:

- в A0 — адрес порта RastPort.
- в A1 — адрес блока данных, задающего бордюр. Формат этого блока будет рассмотрен ниже.
- в D0 — X- координата начальной точки (относительно левого верхнего угла экрана).
- в D1 — Y - координата начальной точки.

РАБОТА С INTUITION

Использовать функцию DrawBorder можно, например, так:

DrawBorder = -108

```
...
borderdraw:                                ;* нарисовать бордюр
    move.l    intbase(PC),A6                ;база intuition.library
    move.l    windowhd(PC),A2              ;идентификатор окна
    move.l    50(A0),A0                    ;адрес порта RastPort
    jsr       DrawBorder(A6)               ;вызываем DrawBorder
    rts                                     ;все
```

Теперь рассмотрим формат блока данных, описывающего отрезки бордюра.

Первые два слова определяют смещение относительно начальной точки бордюра. Чтобы обеспечить попадание всех отрезков внутрь экрана, поместим сюда нули

```
dc.w    0                                ;X — смещение
dc.w    0                                ;Y — смещение
```

Следующие два байта определяют цвета бордюра и фона:

```
dc.b    3                                ;цвет бордюра (красный)
dc.b    0                                ;цвет фона
```

Отрезки бордюра могут выводиться в одном из двух режимов: JAM1 и XOR. В режиме JAM1 отрезки рисуются заданным цветом, "затирая" прежнее содержимое окна, а в режиме XOR пиксели, через которые проходят отрезки бордюра, инвертируются (при этом цвета, заданные в предыдущих байтах, игнорируются). Режим рисования бордюра задается в следующем байте блока данных бордюра:

```
dc.b    0                                ;режим JAM1 (для XOR — 2)
```

При этом режим JAM1 задается кодом 0, а режим XOR — кодом 2.

Следующее поле определяет количество точек (вершин) ломаной, составляющей бордюр. Например, чтобы нарисовать бордюр из двух отрезков, нужно задать три вершины: две концевых и одну угловую. Далее мы определим прямоугольный бордюр, для чего нам понадобятся пять вершин (порядок вывода: 1, 2, 3, 4, 1).

РАБОТА С INTUITION

dc.b 5 ;пять точек

Следующее двойное слово должно содержать указатель на таблицу координат вершин бордюра:

dc.l coord ;указатель на таблицу
;координат

И завершает описываемый блок данных указатель на следующий бордюр (или ноль, если других бордюров нет). Этот указатель используется для связывания независимых бордюров, что может быть полезным, например, при рисовании "объемных" кнопок. Однако для простоты мы будем рассматривать только один бордюр

dc.l 0 ;нет других бордюров

Нам осталось только определить массив координат точек бордюра. Этот массив имеет очевидную структуру

```
coord:      ;координаты вершин
            ;вершина 1 (X = -2, Y = -2)
dc.w      -2,-2
dc.w      80,-2      ;вершина 2
dc.w      80,9       ;вершина 3
dc.w      -2,9       ;вершина 4
dc.w      -2,-2      ;вершина 1
```

Вот и все. Приведем еще раз весь блок данных бордюра:

```
border:
dc.w      0           ;X — смещение
dc.w      0           ;Y — смещение
dc.b      3           ;цвет отрезков
dc.b      0           ;цвет фона
dc.b      0           ;режим вывода (JAM1)
dc.b      5           ;5 вершин
dc.l      coord       ;указатель на массив вершин
dc.l      0           ;нет других бордюров coord:
dc.w      -2,-2
dc.w      80,-2
dc.w      80,9
dc.w      -2,9
dc.w      -2,-2
```

Чтобы нарисовать этот бордюр, вставьте следующий фрагмент кода в программу из предыдущего пункта (перед

РАБОТА С INTUITION

меткой "loop"):

lea	border(PC),A1	:адрес блока данных бордюра
moveq	#20,D0	:X — координата начальной
		:точки
moveq	#80	:Y — координата начальной
		:точки
bsr	borderdraw	:рисуем бордюр

7.9. Gadget'ы.

Переходим к заключительной части раздела, посвященного Intuition. Вам уже должно быть знакомо понятие gadget'a, так как оно не раз встречалось в предыдущих разделах книги. Более точно, gadget системы Intuition — это специальный "символ", с помощью которого пользователь может выполнять различные действия над программой. Примерами системных gadget'ов являются символы закрытия и изменения размеров окон.

Пользовательским программам доступны следующие четыре типа gadget'ов:

- ✓ булевские gadget'ы, которые могут принимать два значения — "YES" или "NO" ("ДА" или "НЕТ"). Переключать такие gadget'ы можно с помощью мыши;
- ✓ строковые (текстовые) gadget'ы, которые предназначены для ввода и редактирования текстовой информации;
- ✓ целочисленные gadget'ы, которые используются для ввода целых чисел в программу;
- ✓ пропорциональные gadget'ы, которые позволяют выбирать значение какого-либо параметра с помощью мыши, передвигая так называемый "движок" (slider).

7.9.1. Булевские gadget'ы.

Начнем с gadget'ов самого простого типа, а именно — с булевских gadget'ов. Примером булевского gadget'a может служить символ закрытия окна. Этот gadget может находиться в одном из двух состояний (нажат, не нажат). Попробуем описать такой gadget в терминах системы Intuition.

Любой gadget задается специальной структурой, со-

РАБОТА С INTUITION

держщей 15 записей. Каждое окно содержит поле, предназначенное для хранения указателя на блок данных gadget'ов. В примерах из предыдущих пунктов мы заполняли эти поля нулями, теперь же мы будем указывать адреса структур gadget'ов, которые должны располагаться в соответствующем окне (или экране).

Блок данных булевского gadget'a имеет следующий формат.

Первое двойное слово используется для хранения указателя на следующий gadget окна. Связанные таким образом gadget'ы располагаются в горизонтальный ряд. Список gadget'ов заканчивается нулем:

gadget1:

dc.l 0 ;больше нет gadget'ов

Следующие два слова определяют положение gadget'a внутри окна. Существует несколько способов задания координат gadget'a, мы рассмотрим самый простой:

dc.w 40 ;X — и

dc.w 50 ;Y — координаты gadget'a

Следующие два слова задают размеры "кнопки" gadget'a. Кнопка gadget'a — это область окна, которая используется Intuition для распознавания gadget'a: если пользователь нажимает левую кнопку мыши и при этом указатель находится внутри этой области, соответствующий gadget активизируется. Заметим, что неправильно подобранный размер изображения (или бордюра) gadget'a может привести к тому, что область реакции на нажатие gadget'a окажется больше (или наоборот, меньше) требуемой.

dc.w 32 ;ширина и

dc.w 13 ;высота кнопки gadget'a

Далее следует слово, биты которого определяют свойства gadget'a. Первые два бита (0 и 1) используются для задания реакции на нажатие gadget'a:

Бит0	Бит1	Значение	Название	Реакция
0	0	0	GADGHCOMP	gadget инвертируется
0	1	1	GADGHBOX	gadget выделяется прямоугольником
1	0	2	GADGHIMAGE	появляется дополнительное

РАБОТА С INTUITION

1	1	3	GAOHNONE	изображение никакой реакции
---	---	---	----------	--------------------------------

Бит 2 определяет внешний вид gadget'a: если этот бит установлен, считается, что gadget представляет собой изображение. Иначе gadget выделяется бордюром.

Следующий бит определяет способ задания координат gadget'a: если этот бит установлен, Y — координата gadget'a отсчитывается от нижней границы окна, иначе — от верхней. Следующий (4-й) бит имеет тот же смысл для X — координаты: если этот бит установлен, то горизонтальная позиция gadget'a вычисляется относительно правой границы окна (относительный способ задания координат).

Заметим, что относительная координата (X или Y) gadget'a должна задаваться отрицательным числом.

Примером gadget'a с абсолютными заданными координатами может служить символ закрытия окна (так как он находится в левом верхнем углу и его положение не зависит от размеров окна). Примером gadget'a с относительно заданными координатами является символ изменения размера (resize gadget).

Ширина и высота gadget'a также могут задаваться относительно правой и нижней границ окна. Биты 5 и 6 определяют способ задания размеров gadget'a по X и Y соответственно.

Бит 7 (маска \$80) определяет, нужно-ли активизировать gadget при открытии окна.

Если бит 8 (маска \$100) установлен, gadget не может быть активизирован.

В качестве примера рассмотрим gadget, представляющий собой изображение, размеры и положение которого заданы относительно левой и верхней границ окна (абсолютные координаты). Для выделения активизированного gadget'a выберем инверсию. Тогда слово, определяющее свойства gadget'a будет выглядеть так:

dc.w	4	;свойства gadget'a
------	---	--------------------

РАБОТА С INTUITION

Следующее слово блока данных gadget'a содержит биты, определяющие функции gadget'a:

Бит	Значение	Название	Функция
0	1	RELVERIFY	gadget активизируется только тогда, когда пользователь отпускает левую кнопку мыши
1	2	GADGIMMEDIATE	gadget активизируется при нажатии левой кнопки мыши
2	4	ENDGADGET	при активизации gadget'a реквестер убирается
3	8	FOLLOWMOUSE	позволяет отслеживать координаты указателя мыши
4	\$10	RIGHTBORDER	размеры бордюра автоматически подстраиваются
5	\$20	LEFTBORDER	под размеры поля gadget'a
6	\$40	TOPBORDER	
7	\$80	BOTTOMBORDER	
8	\$100	TOGGLESELECT	gadget является переключателем для текстовых gadget'ов
9	\$200	STRINGCENTER	производится автоматическая центровка текста
10	\$400	STRINGRIGHT	то же, только текст выравнивается по правой границе (если оба бита очищены, производится выравнивание по левой границе)
11	\$800	LONGINT	gadget используется для ввода целых чисел
12	\$1000	ALTKEYMAP	для текстовых gadget'ов включается режим альтернативного ввода с клавиатуры (ALT)

Для нашего примера выберем флаги TOGGLESELECT и GADGIMMEDIATE:

dc.w \$102 ;флаги

Следующее слово содержит набор флагов, определяющих тип gadget'a.

Бит	Значение	Название	Тип
0	1	BOOLGADGET	gadget является булевым
1	2	GADGET002	
2	4	STRGADGET	задает текстовый (или

РАБОТА С INTUITION

0+1	3	PROPGADGET	целочисленный) gadget задает пропорциональный gadget
Системные gadget'ы:			
4	\$10	SIZING	gadget изменения размера
5	\$20	WDragging	gadget перемещения окна
4+5	\$30	SDragging	gadget перемещения экрана
6	\$40	WUPFRONT	gadget вынесения окна на передний план
6+4	\$50	SUPFRONT	gadget вынесения экрана на передний план
6+5	\$60	WDOWNBACK	gadget перемещения окна на задний план
6+5+4	\$70	SDOWNBACK	gadget перемещения экрана на задний план
7	\$80	CLOSE	символ закрытия окна (close gadget)
Типы:			
12	\$1000	REQGADGET	gadget реквестера
13	\$2000	GZZGADGET	бордюр окна типа GIMMEZEROZERO
14	\$4000	SCRGADGET	gadget экрана
15	\$8000	SYSGADGET	системный gadget

В качестве примера зададим простой булевский gadget:

```
dc.w 1 ;тип gadget'a
```

Далее следует указатель на структуру, определяющую внешний вид gadget'a (изображение или бордюр). Так как мы выбрали изображение для представления gadget'a, поместим сюда его адрес (формат структуры изображения мы подробно рассматривали в предыдущем пункте):

```
dc.l image ;внешний вид gadget'a
```

Следующее поле имеет смысл только в том случае, если установлен флаг GADGHIMAGE (см. выше), и содержит указатель на изображение (или бордюр), которое должно появляться при активизации gadget'a. Если бит GADGHIMAGE очищен (как в нашем примере), это поле игнорируется.

```
dc.l 0 ;нет дополнительного изображения
```

Следующее двойное слово содержит указатель на структуру, определяющую текст для вывода в поле gad-

РАБОТА С INTUITION

get'a. Если текст не нужен, используется нулевой указатель. Однако мы все же определим некоторый текст:

```
dc.l    ggtext    ;текст gadget'a
```

Следующее двойное слово определяет gadget'ы, которые должны быть деактивизированы, как только активизируется данный gadget. Поместим сюда ноль:

```
dc.l    0         ;нет связей
```

Следующее поле используется только для строковых и пропорциональных gadget'ов и содержит указатель на специальную структуру (SpecialInfo), описывающую характеристики gadget'a:

```
dc.l    0         ;так как мы используем булевский
                    ;gadget
```

Следующее слово содержит идентификатор gadget'a:

```
dc.w    1         ;идентификатор gadget'a
```

И завершает блок данных gadget'a пустое слово:

```
dc.l    0         ;не используется
```

Итак, мы получили следующую таблицу:

gadget1:

```
dc.l    0         ;больше нет gadget'ов
dc.w    40        ;X — координата
dc.w    50        ;Y — координата
dc.w    32        ;ширина
dc.w    13        ;высота
dc.w    4         ;флаги
dc.w    $102      ;флаги активизации
dc.w    1         ;тип gadget'a: булевский
dc.l    image     ;изображение gadget'a
dc.l    0         ;нет дополнительных изображений
dc.l    ggtext    ;текст gadget'a
dc.l    0         ;нет связей
dc.l    0         ;нет SpecialInfo
dc.w    1         ;идентификатор gadget'a
dc.l    0         ;зарезервировано
```

В предыдущем пункте мы уже подготовили структуру, определяющую изображение gadget'a (переключатель в положении "OFF"). Осталось только определить текст

РАБОТА С INTUITION

gadget'a:

ggtext:

dc.b	1,0	;цвета
dc.b	1	;режим
align		
dc.w	-8,14	;X и Y координаты
dc.l	0	;шрифт: стандартный
dc.l	swtext	;текст gadget'a
dc.l	0	;больше нет текста swtext:
dc.b	'Switch',0	;текст: "переключатель"
align		

Введите эту программу в компьютер, оттранслируйте и запустите. Вы увидите gadget в виде "переключателя". Нажмите на изображение gadget'a с помощью мыши — gadget выделится инверсией. Нажмите левую кнопку еще раз — gadget вернется в первоначальное состояние.

Поэкспериментируем с содержимым блока данных gadget'a. Измените значение флагов реакции с 4 на 5 — gadget будет выделяться прямоугольником вместо инверсии. Если Вы теперь установите флаг RELVERIFY (бит 0: +1) в слове, определяющем функции gadget'a, включится режим "надежной" активизации: нажмите левую кнопку мыши, подведя указатель к полю gadgeta, и затем, не отпуская кнопку, выведите указатель за пределы gadget'a. Вы увидите, что gadget деактивизируется. Таким образом можно избежать последствий случайного нажатия gadget'a.

А теперь попробуем перевести переключатель в состояние "ON" (включено). Для этого нам понадобится еще одно изображение, адрес которого мы поместим сразу после адреса основного изображения gadget'a. Установив флаг GADHIMAGE в слове, определяющем функции gadget'a, мы включим режим вывода дополнительного изображения при активизации gadget'a.

Вот структура изображения для переключателя в состоянии "ON":

image2:

dc.w	0,0	;смещение
------	-----	-----------

РАБОТА С INTUITION

```

dc.w      32,13          ;размеры
dc.w      1              ;режим: I
dc.l      imgdat,A2      ;биты изображения
dc.b      2,1           ;цвета
dc.l      0              ;больше ничего
imgdata2:  ;изображение переключателя
dc.l      %00000000000000000000000000000000
dc.l      %00000000011100000000000000000000
dc.l      %00000000111100000110100100000000
dc.l      %00000000111100000101010101000000
dc.l      %00000000011100000101010110000000
dc.l      %00000000000111000011101001000000
dc.l      %00000000000011100000000000000000
dc.l      %00000000000001110000000000000000
dc.l      %00000000000011111111110000000000
dc.l      %00000000001111111111100000000000
dc.l      %00000000001111111111100000000000
dc.l      %00000000000110000001100000000000
dc.l      %00000000000000000000000000000000

```

Теперь по изображению gadget'a можно определить, активизирован он или нет.

7.9.2. Текстовые gadget'ы.

Предположим, что нам нужно написать программу для чтения некоторых данных с диска. Для ввода имени файла с клавиатуры нужно, во-первых, вывести подсказку, а во-вторых, вызвать подпрограмму обработки клавиатуры.

Однако существует более простой и элегантный способ, который основывается на использовании текстовых gadget'ов. С помощью текстовых gadget'ов можно легко вводить и редактировать текстовые данные, такие, как имена файлов.

Текстовый gadget — это область окна (или экрана), выделенная с помощью бордюра, внутри которой может осуществляться ввод и редактирования текста. Если при этом текст не умещается в поле ввода, его можно “прокручивать” с помощью клавиш управления курсором.

Особой разновидностью текстовых gadget'ов являются целочисленные gadget'ы. С помощью gadget'ов этого ти-

РАБОТА С INTUITION

па можно вводить и редактировать целые числа, причем перевод строки цифр в двоичное представление осуществляется автоматически.

При редактировании текста можно использовать следующие клавиши:

Клавиши управления курсором (вправо/влево) используются для перемещения курсора в пределах поля ввода, а также для сдвига текстовых строк, длина которых превышает ширину gadget'a.

Клавиши правления курсором + Shift используются для установки курсора в начало/конец текста.

**Клавиша ** используется для удаления символа, на который указывает курсор.

Клавиша <Backspace> используется для удаления символа слева от курсора.

Клавиша <Return> фиксация введенного текста.

Правая <Amiga> + "Q" используется для отмены результатов редактирования (undo).

Более старшие версии Intuition (например, 2.0) используют еще ряд клавиш редактирования (например, Shift + <Backspace>), информацию о которых можно найти в соответствующем справочном руководстве.

Структура блока данных текстовых gadget'ов совпадает со структурой булевских gadget'ов, рассмотренной в предыдущем пункте. Отличие состоит только в том, что, во-первых, тип gadget'a должен быть определен как текстовый (флаг STRGADGET должен быть установлен), и во-вторых, поле SpecialInfo должно содержать указатель на блок данных (strinfo), структуру которого мы рассмотрим ниже.

Первое слово должно содержать адрес области памяти для хранения текста:

strinfo:

dc.l strbuffer ;текстовый буфер

Далее следует адрес буфера отмены результатов редактирования (undo-buffer). Размер этого буфера должен быть не меньше, чем размер буфера текста. Каждый раз

РАБОТА С INTUITION

при активизации gadget'a содержимое основного буфера текста копируется в undo-буфер, а при нажатии комбинации клавиш <Right Amiga>+"Q" восстанавливается из undo-буфера. Один и тот же undo-буфер может использоваться сразу в нескольких gadget'ах, так как в каждый момент времени активизирован может быть только один из них.

dc.l undo ;адрес undo-буфера

Следующее слово определяет максимальное число символов во вводимой строке. При попытке ввести большее количество символов, чем указано в этом слове, экран на короткое время "подсветится", сигнализируя о невозможности дальнейшего ввода. Максимальное число символов текста может быть больше, чем позволяет ширина gadget'a, так как системой Intuition поддерживается скроллинг (прокрутка) текста в окне ввода.

dc.w 10 ;максимальное число символов

Следующее слово определяет номер символа, с которого должен начинаться вывод текста в поле редактирования gadget'a. Поместим сюда ноль, чтобы пользователь мог видеть начало текста:

dc.w 0 ;начальный символ

Следующие пять слов зарезервированы для внутреннего использования, так что их можно инициализировать нулями:

dc.w 0 ;номер символа в undo-буфере

dc.w 0 ;количество символов в буфере текста

dc.w 0 ;количество символов в окне ввода

dc.w 0 ;горизонтальное смещение окна ввода

dc.w 0 ;вертикальное смещение окна ввода

Следующие два слова также используются внутри системы Intuition:

dc.l 0 ;адрес порта RastPort

dc.l 0 ;слово, содержащее вводимое

 ;число (для целочисленных gadget'ов)

И, наконец, последняя запись описываемого блока данных должна содержать указатель на альтернативную

РАБОТА С INTUITION

таблицу клавиатуры (используется, если установлен бит ALTKEYMAP в слове, определяющем функции gadget'a):

```
dc.l    0                ;использовать стандартную
                        ;таблицу клавиатуры
```

Приведем еще раз весь блок данных нашего текстового gadget'a:

strinfo:

```
dc.l    strbuffer        ;адрес буфера текста
dc.l    undo              ;адрес undo-буфера
dc.w    0                 ;позиция курсора
dc.w    10                ;максимальное число символов
dc.w    0                 ;начальный символ
dc.w    0                 ;номер символа в undo-буфере
dc.w    0                 ;количество символов в буфере
dc.w    0                 ;количество видимых символов
dc.w    0                 ;смещение окна ввода по X
dc.w    0                 ;смещение окна ввода по Y
dc.l    0                 ;адрес порта RastPort
dc.l    0                 ;вводимое число (для
                        ;целочисленных gadget'ов)
dc.l    0                 ;стандартная таблица
                        ;клавиатуры
```

Определим буфер текста и undo-буфер:

strbuffer:

```
dc.b    "Hello !",0,0,0
agalign
```

undo:

```
dc.l    0,0,0,0
```

Теперь Вы можете либо изменить структуру gadget'a из предыдущего пункта, либо создать новую, специально для текстового gadget'a, и связать ее с булевым gadget'ом. Мы рекомендуем второй путь, так как "переключатель" из предыдущего пункта Вам еще может понадобиться. Определим текстовый gadget с помощью следующей структуры:

```
gadget1:                ;* структура для текстового gadget'a
dc.l    0                ;больше нет gadget'ов
dc.w    20,80            ;координаты gadget'a
```

РАБОТА С INTUITION

	dc.w	80,10	;размеры gadget'a
	dc.w	0	;флаги
	dc.w	2	;режим активизации
	dc.w	4	;тип gadget'a: текстовый
	dc.l	border	;бордюр
	dc.l	0	;нет изображения
	dc.l	0	;нет текста
	dc.l	0	;нет связей с другими gadget'ами
	dc.l	strinfo	;адрес структуры SpecialInfo
	dc.w	2	;идентификатор gadget'a
	dc.l	0	;не используется
border:			; * блок данных бордюра
	dc.w	0,0	;смещения
	dc.b	3,3	;цвета
	dc.b	0	;режим: JAM1
	dc.b	5	;пять вершин
	dc.l	coord	;координаты вершин
	dc.l	0	;больше нет бордюров
coord:			; * координаты вершин бордюра
	dc.w	-2,-2	;левый верхний угол
	dc.w	80,-2	;правый верхний
	dc.w	80,9	;правый нижний
	dc.w	-2,9	;левый нижний
	dc.w	-2,-2	;снова левый верхний

После запуска программы Вы увидите надпись "Hello!", обведенную красным прямоугольником. Это и есть строковый gadget. Подведите к нему указатель мыши и нажмите левую кнопку, после чего Вы сможете редактировать текст gadget'a. Зафиксировать изменения буфера текста можно либо нажатием клавиши <Return>, либо нажатием левой кнопки мыши за пределами gadget'a.

А теперь измените слово, определяющее флаги активации, на \$802, а также буфер strbuffer на "dc.l 0,0,0,0". После запуска полученной программы Вы сможете вводить только цифры (при попытке ввести любой другой символ экран будет "мигать"). Это сигнализирует о том, что активизированный gadget является целочисленным.

Введите какое-нибудь число, и после завершения

РАБОТА С INTUITION

программы распечатайте содержимое структуры StrInfo. Восьмое слово (двойное) этой структуры как раз будет содержать введенное Вами число.

7.9.3. Пропорциональные gadget'ы.

Вы наверняка знакомы с устройствами, в которых для определения некоторых параметров используются "движки" (sliders). Например, в магнитофонах движки могут использоваться для задания громкости и других характеристик звука. Согласитесь, что такой способ задания параметров в некоторых случаях оказывается намного удобнее, чем обычный ввод чисел. Система Intuition позволяет создавать на экране Amiga "движковые" устройства, управляемые с помощью мыши.

Для этого используются gadget'ы специального типа — пропорциональные. Пропорциональный gadget представляет собой прямоугольную область, внутри которой располагается некоторый символ. Этот символ можно перемещать в пределах gadget'a в горизонтальном и/или вертикальном направлении, "захватив" его мышью. Размеры поля gadget'a могут задаваться относительно размеров окна, так что при изменении размеров окна gadget будет автоматически масштабироваться.

Особенности работы с пропорциональными gadget'ами лучше всего рассматривать на примерах. Предположим, что нам нужно создать простой горизонтальный движок.

Для этого нам понадобится блок данных, описывающий gadget. Формат этого блока совпадает с описанным в предыдущих пунктах, отличия встречаются лишь при интерпретации некоторых параметров. Рассмотрим эти отличия на примере:

gadget2: ;блок данных для пропорционального

;gadget'a

dc.l	0	;больше нет gadget'ов
dc.w	150,30	;координаты и
dc.w	100,10	;размеры gadget'a

РАБОТА С INTUITION

dc.w	4	;флаг: GADGIMAGE
dc.w	2	;активизация: GADGIMMEDIATE
dc.w	3	;тип: пропорциональный gadget
dc.l	mover	;изображение движка
dc.l	0	;нет дополнительного изображения
dc.l	0	;нет текста
dc.l	0	;нет связей
dc.l	propinfo	;адрес структуры PropInfo
dc.w	3	;идентификатор gadget'a
dc.l	0	;не используется

Поместите адрес этого блока (gadget2) в первое слово блока gadget1 из предыдущего пункта, в результате чего наш пропорциональный gadget окажется привязанным к окну.

Указатель mover используется при определении внешнего вида символа движка. Выберем, например, следующее изображение:

mover: ;* внешний вид движка

dc.w	0,0	;нет смещения
dc.w	16,7	;размеры
dc.w	1	;один битплан
dc.l	moverdata	;биты изображения
dc.b	1,0	;цвет: белый
dc.l	0	;больше нет изображений

moverdata: ;изображение движка

dc.w	%0111111111111110
dc.w	%0101111111111010
dc.w	%0101011111101010
dc.w	%0101010110101010
dc.w	%0101011111101010
dc.w	%0101111111111010
dc.w	%0111111111111110

Теперь рассмотрим формат структуры PropInfo, которая, как и StrInfo из предыдущего пункта, описывает особые свойства gadgeta.

Блок данных PropInfo начинается со слова, содержащего набор некоторых управляющих флагов:

РАБОТА С INTUITION

Бит	Значение	Название	Назначение
0	1	AUTOKNOB	движок устанавливается автоматически
1	2	FREEHORIZ	движок может двигаться горизонтально
2	4	FREEVERT	движок может двигаться вертикально
3	8	PROP_BORDERLESS	не использовать системный бордер
8	\$100	KNOB_HIT	этот бит устанавливается при "захвате" движка с помощью мыши (устанавливается системой Intuition)

Нулевой бит (AUTOKNOB) используется при создании простых пропорциональных gadget'ов. Если этот бит установлен, Intuition создает белый движок, размеры которого соответствуют размерам gadget'a, а также диапазону, из которого выбирается значение параметра gadget'a. Например, если Вы используете пропорциональный gadget для представления числа изображенных строк в программе просмотра текста, то размер движка выбирается так, чтобы отношение количества изображенных строк к общему количеству строк в тексте равнялось отношению размера gadget'a к размеру движка. Таким образом, чем больше строк помещается на экране, тем больше будет размер движка.

При использовании таких gadget'ов поле mover блока данных gadget'a должно содержать указатель на специальный буфер размером 8 байт:

buffer:

```
dc.w    0      ;X — координата движка
dc.w    0      ;Y — координата движка
dc.w    0      ;ширина движка
dc.w    0      ;высота движка
```

Мы не будем использовать режим AUTOKNOB всякий раз с ограничениями на внешний вид движка gadget'a, поэтому флаги gadget'a мы определим так:

```
dc.w    2      ;флаги: FREEHORIZ
```

В следующих полях структуры PropInfo содержатся параметры gadget'a, а именно — горизонтальная и вертикальная координаты движка. Нулевое значение соответствует крайнему левому (или верхнему) положению движка, а значение \$FFFF — крайнему правому (или нижнему).

РАБОТА С INTUITION

Именно в этом диапазоне (0-\$FFFF) могут изменяться параметры gadget'a.

dc.w 0,0 ;начальные координаты gadget'a

Далее следуют два слова (HorizBody и VertBody), которые определяют размер движка в режиме AUTOKNOB, либо шаг движка (то есть количество пикселей, на которое перемещается движок, когда пользователь нажимает кнопку мыши справа или слева от него):

dc.w \$FFFF/16 ;горизонтальный шаг: 1/16

dc.w 0 ;нет вертикального перемещения

И завершают блок PropInfo шесть служебных слов, которые инициализируются системой Intuition:

dc.w 0 ;ширина поля gadget'a

dc.w 0 ;высота поля gadget'a

dc.w 0 ;горизонтальный шаг в пикселях

dc.w 0 ;вертикальный шаг в пикселях

dc.w 0 ;левый бордюр

dc.w 0 ;правый бордюр

Таким образом, получаем следующую структуру:

propinfo:

dc.w 2 ;флаги: FREEHORIZ

dc.w 0,0 ;начальные координаты gadget'a

dc.w \$FFFF/16 ;горизонтальный шаг: 1/16

dc.w 0 ;нет вертикального перемещения

dc.w 0 ;ширина поля gadget'a

dc.w 0 ;высота поля gadget'a

dc.w 0 ;горизонтальный шаг в пикселях

dc.w 0 ;вертикальный шаг в пикселях

dc.w 0 ;левый бордюр

dc.w 0 ;правый бордюр

Попробуйте изменить эту структуру для определения вертикального gadget'a.

7.10. Пример программы.

В этом пункте мы приведем пример законченной программы, демонстрирующей рассмотренные средства Intuition по созданию графических интерфейсов.

;7_Intuition.asm

;** Demo-Program for working with intuition **

РАБОТА С INTUITION

```
movescreen    = -162
openscreen    = -198
closescreen   = -66
openwindow    = -204
closewindow   = -72
autorequest   = -348
setmenustrip  = -264
clearmenustrip = -54
printitext    = -216
drawimage     = -114
drawborder    = -108
displaybeep   = -96
closelibrary  = -414
openlib       = -552
execbase      = 4
getmsg        = -372
joy2          = $dff00c
fire          = $bfe001
```

run:

```
bsr    openint
bsr    scropen
bsr    windopen
bsr    setmenu
bsr    print
lea     border(PC),A1
moveq   #22,D0
moveq   #30,D1
bsr    borderdraw
bsr    draw
bsr    request
```

loop:

```
move.l   execbase,A6
move.l   windowhd(PC),A0
move.l   86(A0),A0      ;пользовательский порт
jsr      GetMsg(A6)
tst.l    D0
beq.s    loop           ;события не было
move.l   D0,A0
move.l   $16(A0),msg     ;событие
move.l   msg(PC),D6      ;для тестирования
```

РАБОТА С INTUITION

```

    move.l    D6,D7
    lsr       #8,D7
    lsr       #3,D7
    moveq     #0,D5
    roxr      #1,D6
    roxl      #1,D5
    and.l     #$7F,D6
    cmp       #$7F,D6
    beq.s     loop
    lsr       #4,D6
    cmp       #1,D6
    bne.s     nol
    move.l     intbase(PC),A6
    move.l     screenhd(PC),A0
    jsr       displaybeep(A6)

nol:
    cmp       #0,D6
    bne.s     loop

ende:
    bsr       clearmenu
    bsr       windclose
    bsr       scrclose
    bsr       closeint
    rts

openint:                                           ;* открыть и инициализировать
                                                    ;* Intuition
    move.l     ExecBase,A6
    lea        IntName(PC),A1
    moveq      #0,D0
    jsr        OpenLib(A6)
    move.l     D0,intbase
    rts

closeint:                                           ;* закрыть Intuition
    move.l     ExecBase,A6
    move.l     intbase(PC),A1
    jsr        CloseLib(A6)
    rts

scropen:                                           ;* открыть экран
    move.l     intbase(PC),A6
    lea        screen_defs(PC),A0

```

РАБОТА С INTUITION

```

jsr      OpenScreen(A6)
move.l   D0,screenhd
rts

scrclose:                                ;* закрыть экран
move.l   intbase(PC),A6
move.l   screenhd(PC),A0
jsr      CloseScreen(A6)
rts

scrmove:                                    ;передвинуть экран
move.l   intbase(PC),A6
move.l   screenhd(PC),A0
moveq.l  #0,D0
jsr      MoveScreen(A6)
rts

windopen:                                ;* открываем окно
move.l   intbase(PC),A6
lea      windowdefs(PC),A0
jsr      OpenWindow(A6)
move.l   D0>windowhd
rts

windowclose:                              ;* закрываем окно
move.l   intbase(PC),A6
move.l   windowhd(PC),A0
jsr      CloseWindow(A6)
rts

request:                                   ;* выводим реквестер
move.l   windowhd(PC),A0
lea      btext(PC),A1
lea      ltext(PC),A2
lea      rtext(PC),A3
moveq.l  #0,D0
moveq.l  #0,D1
move.l   #180,D2
moveq.l  #80,D3
move.l   intbase(PC),A6
jsr      autorequest(A6)
rts

setmenu:                                   ;* создает и инициализирует блоки
                                           ;* данных меню
lea      mentab(PC),A0

```

РАБОТА С INTUITION

```

    lea      menu(PC),A1
    moveq    #10,D1
menuloop:
    moveq    #0,D2
    move.l   A1,A2
    tst.l    (A0)
    beq      setmenu1
    clr.l    (A1)+
    move     D1,(A1)+
    add.l    #70,D1
    move.l   #50,(A1)+
    move.l   #$A0001,(A1)+
    move.l   (A0)+,(A1)+
    lea      12(A1),A3
    move.l   A3,(A1)+
    clr.l    (A1)+
    clr.l    (A1)+
itemloop:
    tst.l    (A0)
    beq      menuend
    lea      54(A1),A3
    move.l   A3,(A1)+
    move.l   D2,(A1)+
    add      #10,D2
    move.l   #$5A000A,(A1)+
    move     #$52,(A1)+
    clr.l    (A1)+
    lea      16(A1),A3
    move.l   A3,(A1)+
    clr.l    (A1)+
    clr.l    (A1)+
    clr.l    (A1)+
    move     #1,(A1)+
    clr      (A1)+
    move.l   #$50003,(A1)+
    clr.l    (A1)+
    move.l   (A0)+,(A1)+
    clr.l    (A1)+
    bra.s    itemloop
menuend:

```

РАБОТА С INTUITION

```
clr.l    -54(A1)
tst.l    (A0)+
tst.l    (A0)
beq.s    setmenu1
move.l    A1,(A2)
bra      menuloop

setmenu1:                                ;* создать меню
move.l    intbase(PC),A6
move.l    windowhd(PC),A0
lea       menu(PC),A1
jsr       SetMenuStrip(A6)
rts

clearmenu:                               ;* уничтожаем меню
move.l    intbase(PC),A6
move.l    windowhd(PC),A0
lea       menu(PC),A1
jsr       clearmenustrip(A6)
rts

print:                                       ;* печатаем текст
move.l    intbase(PC),A6
move.l    windowhd(PC),A0
move.l    50(A0),A0
lea       ggtext(PC),A1
moveq.l   #30,D0
moveq.l   #60,D1
jsr       printitext(A6)
rts

draw:
move.l    intbase(PC),A6
move.l    windowhd(PC),A0
move.l    50(A0),A0
lea       image(PC),A0
move.l    #200,D0
move.l    #100,D1
jsr       drawimage(A6)
rts

borderdraw:                               ;* нарисовать бордюр
move.l    intbase(PC),A6
move.l    windowhd(PC),A2
move.l    50(A0),A0
```

ЛЕСНА: ПРОГРАМИРАЊЕ НА АСЕМБЛЕРЕ.

РАБОТА С INTUITION

```
      jsr      DrawBorder(A6)
      rts

screen_defs:
      dc.w      0,0
      dc.w      640,200
      dc.w      4
      dc.b      0
      dc.b      1
      dc.w      $800
      dc.w      15
      dc.l      0
      dc.l      titel
      dc.l      0
      dc.l      0

windowdef:
      dc.w      10,20
      dc.w      300,150
      dc.b      0,1
      dc.l      $300
      dc.l      $100F
      dc.l      gadget
      dc.l      0
      dc.l      windname

screenhd: dc.l      0
          dc.l      0
          dc.w      200,40,600,200
          dc.w      $FF

btext:
      dc.b      3,3
      dc.b      0
      align
      dc.w      10,10
      dc.l      0
      dc.l      bodytxt
      dc.l      0
bodytxt: dc.b      "Requester text",0
          align

ltext:
      dc.b      3,1
      dc.b      0
```


РАБОТА С INTUITION

```

align
dc.w      5,3
dc.l      0
dc.l      lefttxt
dc.l      0
lefttxt:  dc.b      "left",0
align
rtext:
dc.b      0,1
dc.b      0
align
dc.w      5,3
dc.l      0
dc.l      righttxt
dc.l      0
righttxt: dc.b      "right",0
align
titel:    dc.b      "User Screen",0
windname:
dc.b      "Window-title",0
align
windowwhd:
dc.l      0
intbase:  dc.l      0
intname:  dc.b      "intuition.library",0
align
msg:      dc.l      0
mentab:
dc.l      menu1
dc.l      mp11,mp12,mp13,mp14,mp15,mp16,mp17,mp18,mp19,0
dc.l      menu2
dc.l      mp21,mp22,mp23,0
dc.l      menu3
dc.l      mp32,mp32,0
dc.l      menu4,mp41,0
dc.l      0
menu1:    dc.b      "Menu 1",0
mp11:     dc.b      "Point 11",0
mp12:     dc.b      "Point 12",0
mp13:     dc.b      "Point 13",0

```

РАБОТА С INTUITION

```

mp14:  dc.b    "Point 14",0
mp15:  dc.b    "Point 15",0
mp16:  dc.b    "Point 16",0
mp17:  dc.b    "Point 17",0
mp18:  dc.b    "Point 18",0
mp19:  dc.b    "Point 19",0
menu2: dc.b    "Menu 2",0
mp21:  dc.b    "End !",0
mp22:  dc.b    "Beep",0
mp23:  dc.b    "Point 23",0
menu3: dc.b    "Menu3",0
mp31:  dc.b    "Point 31",0
mp32:  dc.b    "Point 32",0
menu4: dc.b    "Menu 4",0
mp41:  dc.b    "Point 41",0
      align

gadget:
      dc.l    gadgetl
      dc.w    20,80,80,10
      dc.w    0
      dc.w    $2
      dc.w    4
      dc.l    border
      dc.l    0
      dc.l    0
      dc.l    0
      dc.l    strinfo
      dc.w    2
      dc.l    0

border:
      dc.w    0,0
      dc.b    1,0,0
      dc.b    5
      dc.l    koord
      dc.l    0

koord:
      dc.w    -2,-2,80,-2,80,9,-2,9,-2,-2

strinfo:
      dc.l    strbuffer
      dc.l    undo
    
```

РАБОТА С INTUITION

```
dc.w 0
dc.w 10
dc.w 0
dc.w 0,0,0,0
dc.l 0,0,0
strbuffer: dc.b "Hello !",0,0,0
undo: dc.l 0,0,0
align
gadget1:
dc.l gadget2
dc.w 40,50,32,13
dc.w 6
dc.w $103
dc.w 1
dc.l image
dc.l image2
dc.l ggtext
dc.l 0
dc.l 0
dc.w 1
dc.l 0
ggtext:
dc.b 1,0,1
align
dc.w -8,14
dc.l 0
dc.l swtext
dc.l 0
swtext: dc.b "Switch",0
align
image:
dc.w 0,0
dc.w 32,13
dc.w 1
dc.l imgdata
dc.b 2,1
dc.l 0
image2:
dc.w 0,0
dc.w 32,13
```

РАБОТА С INTUITION

```

dc.w      1
dc.l      imgdata2
dc.b      2,1
dc.l      0

imgdata:
dc.l      %00000000000000000000000000000000
dc.l      %00000000001110000000000000000000
dc.l      %00000000111110000011101001000000
dc.l      %00000000111110000010101101000000
dc.l      %00000000011110000010101011000000
dc.l      %00000000000111000011101001000000
dc.l      %00000000000011100000000000000000
dc.l      %00000000000011100000000000000000
dc.l      %00000000000011111111110000000000
dc.l      %00000000001111111111100000000000
dc.l      %00000000001111111111100000000000
dc.l      %00000000000110000001100000000000
dc.l      %00000000000000000000000000000000

imgdata2:
dc.l      %00000000000000000000000000000000
dc.l      %000000000000000000000000111000000000
dc.l      %0001110111011100000111110000000000
dc.l      %0001010100010000000111110000000000
dc.l      %0001010110011000000111100000000000
dc.l      %00011101000100000111000000000000
dc.l      %0000000000000000001110000000000000
dc.l      %0000000000000000011100000000000000
dc.l      %0000000000000000111000000000000000
dc.l      %0000000000011111111110000000000000
dc.l      %0000000000111111111110000000000000
dc.l      %0000000000111111111110000000000000
dc.l      %0000000000011000000110000000000000
dc.l      %00000000000000000000000000000000

gadget2:
dc.l      0
dc.w      150,30,100,50
dc.w      5
dc.w      2
dc.w      3
dc.l      mover
dc.l      0,0,0
    
```

РАБОТА С INTUITION

```

        dc.l    specinfo
        dc.w    3
        dc.l    0
specinfo:
        dc.w    6
        dc.w    0,0
        dc.w    $FFFF/10,$FFFF/5
        dc.w    0,0,0,0,0
mover:
        dc.w    0,0,16,7
        dc.w    1
        dc.l    moverdata
        dc.b    1,0
        dc.l    0
moverdata:
        dc.w    %0111111111111110
        dc.w    %0101111111111010
        dc.w    %0101011111101010
        dc.w    %0101010110101010
        dc.w    %010101111101010
        dc.w    %010111111111010
        dc.w    %01111111111110
menu:   blk.w   500
        end
    
```

ДОПОЛНЕНИЯ**Раздел 8. ДОПОЛНЕНИЯ.**

Итак, мы рассмотрели все основные аспекты машинного программирования на Amiga. В этом (завершающем) разделе мы подробнее остановимся на некоторых дополнительных вопросах, касающихся программирования MC68000. А именно, мы поговорим о работе в режиме супервизора и об использовании исключений.

8.1. Режим супервизора.

Как Вы уже знаете, процессор MC68000 (и старше) может работать в одном из двух режимов: пользователя (user mode) и супервизора (supervisor mode) (см. раздел 2). При программировании некоторых специфических системных задач может возникнуть потребность переключения из одного режима в другой (например, для доступа к регистрам статуса необходимо перейти в режим супервизора).

Как же осуществляется переключение режимов процессора?

Очень просто. Операционная система Amiga содержит функцию, с помощью которой можно перейти в режим супервизора. Эта функция называется SuperState (смещение -150 в exec.library) и не требует никаких параметров:

ExecBase = 4

SuperState = -150

```
...
move.l  ExecBase,A6          ;базовый адрес exec.library
jsr     SuperState(A6)       ;переходим в режим супервизора
move.l  D0,savesp            ;сохраняем точку возврата
...
savesp: blk.l  1              ;место для хранения точки
                               ;возврата
```

ДОПОЛНЕНИЯ

После вызова SuperState в D0 возвращается значение системного указателя стека (SSP), а текущий указатель стека (SP) устанавливается на пользовательский стек (USP). Это дает возможность работать с пользовательским стеком в режиме супервизора.

Для возврата в режим пользователя используется функция UserState (смещение -156 в exec.library), единственным параметром которой является значение системного указателя стека, возвращенное функцией SuperState:

UserState = -156

```
...  
move.l    execbase,A6    ;базовый адрес exec.library  
move.l    savesp(PC),D0  ;восстанавливаем стек (SSP)  
jsr       UserState(A6)  ;возврат в режим пользователя
```

Таким образом можно писать подпрограммы, которые будут выполняться в режиме супервизора. Для этого нужно: вызвать SuperState, сохранить возвращенный указатель стека, выполнить нужные действия, вызвать UserState и выполнить команду RTS.

Возникает вопрос: как работает команда SuperState? Каким образом операционная система переходит в режим супервизора? Опишем этот механизм:

Подпрограмма SuperState делает попытку обращения к регистрам статуса. В результате возникает исключение (по вектору \$20), и происходит переход на подпрограмму-обработчик в режиме супервизора. Эта подпрограмма выясняет, что исключение вызвано обращением к регистрам статуса из функции SuperState, и осуществляет возврат без перехода в режим пользователя.

8.2. Программирование исключений.

Механизм исключений, рассмотренный в разделе 2, предоставляет широкие возможности по отслеживанию ошибочных ситуаций, которые могут возникнуть в программе. Процессор MC680x0 имеет следующий набор исключений:

ДОПОЛНЕНИЯ

Номер	Адрес	Причина
2	\$008	ошибка шины
3	\$00C	ошибка адресации
4	\$010	неправильная команда
5	\$014	деление на ноль
6	\$018	команда CHK
7	\$01C	команда TRAPV
8	\$020	нарушение привелегий
9	\$024	трассировка
10	\$028	эмуляция команд Axxx
11	\$02C	эмуляция команд Fxxx
	\$030-\$03B	зарезервировано
15	\$03C	неинициализированное прерывание
	\$040-\$05F	зарезервировано
24	\$060	неавторизованное прерывание
25-31	\$064-\$07F	прерывания уровней 1-7
32-47	\$080-\$0BF	команды TRAP
	\$0C0-\$0FF	зарезервировано
64-255	\$100-\$3FF	пользовательские вектора

Остановимся подробнее на командах TRAP. Существует 16 различных команд TRAP (коды — от 0 до 15), с каждой из которых связана своя подпрограмма-обработчик. Так, при выполнении команды TRAP #0, процессор переходит в режим супервизора и приступает к выполнению подпрограммы, адрес которой записан в ячейке \$80. Эта подпрограмма должна завершаться командой RTE.

Команды TRAP не используются операционной системой Amiga (в отличие от операционной системы TOS компьютеров ATARI ST, где команды TRAP служат для вызова системных функций), поэтому их можно использовать в программах пользователя.

Напишем собственный обработчик (диспетчер) исключений TRAP. Тестовая программа будет состоять из следующих частей:

1. Инициализация векторов TRAP 2. Подпрограмма обработки исключений TRAP 3. Основная часть, содержащая

ДОПОЛНЕНИЯ

команды TRAP

Первая часть содержит две команды:

```
init:      move.l   #trap0,$80      ;установить вектор для TRAP #0
           rts
```

Для написания подпрограммы trap0 воспользуемся примером, приведенным в разделе 5 (воспроизведение короткого звукового сигнала). Изменим команду RTS на RTE, а также модифицируем эту подпрограмму для воспроизведения звука произвольной длительности:

```
;** Beep tone production after a TRAP #0 **
ctlw      = $DFF096 ;управление DMA
c0thi     = $DFF0A0 ;старший адрес таблицы звука
c0tlo     = c0thi+2 ;младший адрес таблицы звука
c0tl      = c0thi+4 ;размер таблицы
c0per     = c0thi+6 ;частота
c0vol     = c0thi+8 ;громкость

trap0:
           move.l   #table,c0thi    ;* воспроизвести звуковой сигнал
           move     #4,c0tl         ;начало таблицы
           move     #300,c0per      ;размер таблицы
           move     #40,c0vol       ;частота
           move     #$8201,ctlw     ;громкость
           move     #1,c0tl         ;старт DMA

loop:
           subq.l   #1,D0           ;счетчик — 1
           bne.s    loop           ;цикл (задержка) still
           move     #1,ctlw         ;выключаем звук
           rte        ;возврат из обработчика исключений

table:
           dc.b     -40,-70,-40,0,40,70,40,0 ;таблица звука
```

Убедитесь, что таблица table находится в chip-памяти, иначе чип Paula не будет иметь доступа к звуковым данным.

И, наконец, тестовая часть нашей программы будет состоять из трех команд:

```
test:
           move.l   #$2FFFF,D0      ;длительность звука
           trap     #0              ;возбуждаем исключение
           rts
```

ДОПОЛНЕНИЯ

Введите полученную программу в компьютер, оттранслируйте и запустите подпрограмму инициализации (init). Ничего не произойдет.

Теперь запустите тестовую часть (test), и Вы услышите звуковой сигнал длительностью около секунды.

Заметим, что при повторном ассемблировании адрес подпрограммы `trap0` может измениться, поэтому перед выполнением тестовой части не забывайте вызывать подпрограмму `init`.

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

Раздел 9. AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

Книга, которую Вы прочитали, ориентирована на ранние модели Amiga: A500, A1000 и A2000. Впоследствии номенклатура выпускаемых Амиг постоянно менялась и расширялась. Появились модели A3000, A600, A4000, A1200. Сейчас (1996-97 годы) основными являются модели A4000 и A1200. Немецкая фирма Phase 5 digital products готовит к выпуску новое поколение "AMIGA-клонов" — так называемый A\BOX, имеющий революционные отличия внутренней архитектуры. Теперь перейдем к изменениям "изнутри".

Описываемые в книге "потроха" Амиги — это ни что иное, как чипсет, ныне называемый OCS (Original ChipSet). Несколько позже (в 1990 году) был выпущен ECS (Enhanced ChipSet), в 1992 году за ним последовал AGA (Advanced Graphics Architecture). Все эти чипсеты программно совместимы "снизу вверх": программа, корректно написанная под OCS, будет нормально работать на ECS и AGA. Если Вы уже некоторое время работаете на современной Амиге, то наверняка обратили внимание, что далеко не все старые программы работают нормально, и при запуске с дискет какой-нибудь старой игрушки, частенько приходится нажимать две кнопки мышки и "мудрить" в bootmenu, не говоря уже о запуске множества утилит типа runit, tude и т.п., загрузке kickstart 1.3 и отключении акселератора.

Даже несмотря на все эти ухищрения, программы иногда так и не удастся запустить. Некоторые программы (чаще всего это demo, например довольно симпатичная "Made in Croatia" группы BINARY) работают только до оп-

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

ределенного момента. Многие программы работают не так, как задумывалось (например, анимация в начале demo "Desert Dream" работает несколько быстрее, чем следовало бы, что приводит к довольно забавным эффектам).

Используя определенные приемы, всех этих неприятностей можно избежать. Когда Вы пишете свою программу, особенно работающую с "железом" напрямую, первым делом обязательно проверьте ее работоспособность на всех доступных конфигурациях. AMIGA — это не БК, которая существует всего в двух-трех фиксированных конфигурациях, имеет один-единственный процессор, раз и навсегда фиксированную структуру видеопамати, одну-единственную версию ПЗУ для каждой конфигурации и так далее (кстати, замечу специально для бывших и нынешних "обитателей" БК: архитектура использованного на БК процессора DEC PDP-11 в какой-то мере послужила основой для архитектуры процессоров Motorola 680x0, и они довольно похожи). И если такие подходы на БК, как вызов процедур из ПЗУ по абсолютным адресам, прямая работа с видеопамтью и т.п. проблем не вызывает и является вполне "стандартным" подходом, то различных вариаций "железных" конфигураций Амиги (в том числе во многом и несовместимых между собой) может быть множество. И между ними есть только два связующих звена: чипсет и операционная система. Более того, чипсет тоже не так постоянен и неизменен, как казалось бы. Поэтому имеет смысл рассмотреть то, что изменяется.

Процессор.

В былые времена на Амиге применялся только один процессор — 68000. С тех пор у нас в руках оказалась вся "гамма" процессоров этой серии — 68010, 68020, 68030, 68040 и 68060. В общем, все перечисленные процессоры тоже совместимы "снизу вверх", но есть и некоторые тонкости.

Процессоры 68000 и 68020 имеют 24-битную шину адреса, а это означает, что старший байт адреса попросту игнорируется. Хранение в этом байте "левых" значений ав-

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

Автоматически вызовет неработоспособность программы на старших моделях процессоров. В программах иногда встречался даже такой вариант: под хранение адреса в памяти отводилось всего три байта, а старший, в лучшем случае, просто обнулялся. Очевидно, такая программа (опять же в лучшем случае) на более современной конфигурации будет работать только в младших 16 мегабайтах адресного пространства (а fast-память частенько находится за этим пределом). Программа же, которая хранит в старшем байте неведомо что, на старших процессорах будет вызывать сбой по ошибке шины (обращение к несуществующему устройству).

Конечно, самое главное и существенное различие — это система команд процессора. Основное “пополнение” системы команд произошло на 68020: процессор “обзавелся” новыми методами адресации, операциями с битовыми полями и т.п. (см. таблицы — приложения 3 и 4). Очевидно, что программа, использующая эти нововведения, не будет работать на 68000.

Самый простой выход — это писать программу чисто в кодах 68000: такая программа будет работать на всей гамме процессоров. С другой стороны, опять же очевидно, что нововведения 68020+ имели целью увеличить производительность и эффективность кода. Если требуется максимальная производительность, обычно выпускается несколько версий программы: под 68000, 68020 и 68040. Первая предназначена для 68000 и 68010, будет работать и на следующих модификациях, но медленнее, чем работала бы в их “родных” кодах. Версии под 68020 и 030 — как правило, одно и то же. Версии под 68040 также обычно рассчитаны и на 68060. Кроме того, существуют версии специально под 68060. Наиболее “продвинутые” программы даже не имеют нескольких версий, а самостоятельно определяют тип процессора и используют для разных процессоров разные критичные по быстродействию участки кода.

Немного поясним такой момент, как кэш инструкций

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

(instruction cache) процессора. В 68000 он не был реализован, поэтому этот момент в книге не описан. Начиная с 68020, процессор имеет кэш инструкций — встроенный блок памяти, в который помещаются коды исполняемых операций, и при частом выполнении одних и тех же инструкций, процессор считывает их коды не из памяти, а из кэша. Благодаря этому быстродействие значительно возрастает: скорость считывания из кэша значительно превышает скорость памяти.

Однако, и здесь есть свои “подводные камни”: старые программы (в основном игрушки и demo) для достижения большой производительности частенько грешили, так называемой, самомодификацией кода, т.е. изменением участков кода программы самой программой. Если изменяется тот участок кода, который уже лежит в кэше — в самом кэше он не будет изменен, и программы начинают работать не так, как задумывалось (если хотите понаблюдать, что же выйдет — запустите на Амиге с 68020 и более старшим процессором известную демку “STATE OF THE ART”, не выключив кэш). Последствия такого подхода могут быть самыми разными — от довольно забавных (как в упомянутом случае) до катастрофических (полной неработоспособности программы на более современном процессоре). Конечно, можно будет пользоваться такой программой, выключая кэш при ее запуске, но при этом реальное быстродействие 68020 и более старших процессоров резко упадет, да и в многозадачке вряд ли удастся сделать раздельное состояние кэша для разных процессоров — то есть упадет и скорость работы всех “параллельно” идущих задач.

В самом крайнем случае, если уж хочется “выжать все” из своей 68000 машины — сделайте распознавание типа процессора, и в случае отличающегося от 68000, исполняйте другой участок кода, не использующий самомодификацию.

Кэш инструкций имеет еще одно свойство. На кэшированном процессоре практически не имеют смысла многие

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ

подходы к оптимизации, обычно популярные на старых 8- и 16-битных машинах типа SPECTRUM и БК, заключающиеся в уменьшении "любой ценой" количества операций и занимаемых процедурой тактов. Поскольку на упомянутых машинах код выполнялся из той же самой памяти, что и операции — это действительно имело смысл, как, впрочем, и на 68000 машинах (в приложении 6 приведены методы расчета времени выполнения инструкций 68000).

Начиная с 68020 чаще всего имеет место ситуация, когда память работает значительно медленнее кэша, и если выполняемый фрагмент программы, в основном, занимается обменом с памятью, то основной смысл начинает иметь не количество инструкций, а скорость памяти — например, как ни уменьшать количество операций в процедуре заполнения кодом памяти, быстрее чем запись в память она работать не будет.

Кроме того, циклы следует реализовывать так, чтобы они по возможности целиком умещались в кэш — то есть в процессе работы цикла не уместившиеся в кэше инструкции не подчитывались бы из памяти. Размер кэша инструкций в 68020 и 68030 — 64 двойных слова (256 байт). Начиная с 68030 у процессора появился еще и кэш данных (тоже 256 байт). Теперь при считывании достаточно небольшой группы адресов данные могут "улететь" в кэш и считывание больше не будет замедляться скоростью работы с памятью. У 68040 оба кэша увеличены до 4096 байт.

Теперь несколько слов о таком явлении, как "узкое место" в программе. Если бы программа представляла собой линейный участок кода без условных ветвлений и циклов, это понятие вряд ли существовало бы. Однако абсолютное большинство программ содержит множество участков, которые за все время работы программы выполняются более чем один раз. Грубо говоря, 20% кода выполняются 80% времени (числа взяты "с потолка", но все равно достаточно точно отражают реальное явление). Соответственно, оптимизация по быстродействию 80% кода практически ни-

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

чего не даст — этот участок кода выполнится всего один-два раза (например, начальная инициализация программы). А оптимизация 20% — резко ускорит программу, потому что эти 20% практически все время и работают. Бывают случаи, когда почти 100% времени работы программы — это выполнение одного и того же цикла — например, так работают музыкальные “плееры”. Что же нужно сделать? Нужно выявить именно этот, наиболее часто исполняющийся код, и если быстроедействие программы критично — имеет смысл переписать в код нужного процессора и хорошо прооптимизировать именно этот участок программы, полностью “уложить” его в кэш, минимизировать работу с памятью и перенести основную нагрузку на регистры.

Никакого смысла в предельной оптимизации и подгонки под заданный процессор всего остального кода обычно нет — какая Вам разница, будет окошко открываться 1/10 секунды или 1/5 секунды?...

Из всего стройного ряда 680х0 несколько “выбиваются” 68020 и 68060. Первый имеет специальные функции работы с модулями, присущие только 68020 (в других процессорах ряда, ни до, ни после, эти функции не были реализованы). Поскольку, вероятно, специалисты фирмы MOTOROLA посчитали нецелесообразным их использование, далее мы не будем заострять на них внимание (что советуем и Вам — конечно, можно поэкспериментировать с операциями CALLM и RTM, но стоит ли? Ведь иначе как на 68020 такая программа работать не будет...). Второй, 68060, имеет весьма серьезные отличия от всего ряда. Этот процессор, скажем так, представляет собой “венец” CISC-технологии, из нее, в данном случае, было выжато все, что можно, и в результате мы имеем высокопроизводительный процессор, выполняющий на тактовой частоте 50 МГц до 200 миллионов операций типа регистр-регистр в секунду.

Очевидно, что за все хорошее нужно платить, что происходит и в данном случае. За счет оптимизации быстрогодействия 68060 лишен некоторых операций — полный их

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

список приведен в приложении 5.

Имеет смысл упомянуть и работу с прерываниями. В книге неоднократно упоминается такой момент, как прямая запись значений в векторы прерываний. Начиная с процессора 68010, появился новый управляющий регистр VBR — Vector Base Register, который является указателем на начало размещения векторов прерываний в памяти. Поскольку с нулевого адреса располагается достаточно медленная chip-память, имеет смысл перенести начало векторов в fast-память, записав в VBR новое значение — это заметно уменьшит расходы времени при выполнении прерываний. В таком случае (а такую функцию сейчас имеют многие сервисные утилиты) занесение значений в таблицу векторов, находящуюся с нуля, не будет иметь никакого смысла. Большинство игрушек и demo при VBR не равно нулю просто не работают. Чтобы компенсировать этот эффект, нужно перед установкой векторов считать VBR и оперировать уже новым адресом. Считывание VBR производится командой `MOVEC VBR,Rn` в режиме супервизора.

Память.

Память на Амиге имеет два типа — chip и fast. Исторически изначально на большинстве Амиг стояла только chip-память. Программисты, работающие на таких машинах, должны обязательно иметь представление, каковы отличия конфигурации с fast-памятью, чтобы программы работали не только на их конфигурациях (попытка использовать блиттер и коппер в fast-памяти ни к чему, кроме забавных “глюков” и зависаний, не приведет...) — для этого всегда отдавайте себе отчет, какую память нужно выделить программе для тех или иных целей, и в какую память грузится исполняемый код при запуске.

Кроме того, расположение fast-памяти на разных конфигурациях может быть различно. Не исключен вариант и нескольких не связанных последовательно участков fast-памяти. Операционная система “знает” про все эти тонкости, но, тем не менее, их часто нужно учитывать. Кон-

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

кретный пример — если у Вас имеется два физически не-связанных участка fast-памяти по 8 мегабайт каждый (свойство конкретной организации “железа” конкретной модели машины или акселератора), то программу, требующую 10 непрерывных мегабайт, запустить не удастся. Из этого можно сделать вывод — по возможности не захватывайте один большой участок памяти, а используйте несколько маленьких, разбивая нужную для работы память на столько мелких кусочков (где действительно требуется непрерывная память), на сколько возможно.

Еще один тип памяти, появившийся только с появлением MMU (Memory Management Unit — устройство управления памятью) — это виртуальная память. MMU управляет страничной памятью, позволяя разбить “виртуальную” память (то есть ту память, которая “видна” процессору) на страницы заданного размера, автоматически переадресовывая их на совершенно другие адреса физической памяти, или вызывать прерывание при обращении к некоторым страницам. При установке пакетов типа VMM можно воспользоваться этой возможностью MMU, как бы расширяя память машины и образуя файл подкачки на диске, где, собственно, и хранится эта “расширенная” память. Такая организация памяти позволяет увеличить ее объем, несколько снизив оперативность работы с ней — то есть в результате работы такой системы память становится не всегда доступной “сразу”. К чему это может привести: программа, рассчитанная на мгновенную реакцию памяти, может работать не так, как надо (“спотыкающийся” звук и т.д.). Более подробно эти вопросы освещены в описаниях соответствующих пакетов.

Звук.

Все ныне существующие чипсеты Амиги совместимы между собой по части звука — звуком с 1985 года на Амиге занимается один и тот же чип — PAULA. Видимо, исходя из стандартности и неизменности звукового “железа”, разработчики ОС Амиги не потрудились сделать не-

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

сколько вызовов, делающих использование звука независимым от аппаратной части машины, и функции ОС сводятся к управлению принадлежностью разных аудио-каналов разным задачам.

Большинство программ, работающих со звуком, напрямую обращаются к чипсету. В настоящее время имеется несколько звуковых карт разных производителей, предлагающих более широкие возможности, нежели чипсет (например, аппаратное микширование более чем четырех каналов, аудио-вход, 16-битный звук и т.п.), и до недавнего момента под каждую из них существовали свои собственные пакеты, а некоторые универсальные пакеты обработки звука предлагали возможность использования одной или нескольких карт.

Не так давно на роль стандарта де-факто стала предлагаться система АНІ (Audio Hardware Independent), предоставляющая возможность программирования звука независимо от методов работы с конкретным "железом". Описание системы АНІ не входит в рамки этой книги; система еще достаточно нова и, судя по всему, будет динамично развиваться. В дистрибутив системы входит подробное описание ее вызовов для разработчиков программного обеспечения.

Кроме того, звуковой чип Амиги имеет некоторые возможности, о которых не подозревали даже... сами разработчики Амиги (воистину, AMIGA не менее неисчерпаема, чем атом...)! Это тот самый 14-битный звук, поддержка которого в последнее время появилась практически во всех "плеерах". Работает он предельно просто: четыре физических канала объединяются по два, образуя, соответственно, левый и правый аудиоканал. На одном канале в паре ставится максимальная громкость, на другом — минимальная. В результате один из каналов хранит 8 младших бит из 14, а шесть старших бит второго канала хранят шесть старших бит (младшие два бита "пересекаются" по назначению со старшими двумя битами из другого канала пары). В ре-

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

зультате, качество звука получается значительно лучше, нежели при использовании только восьми бит.

Bugeo.

Как упоминалось ранее, имеющиеся на данный момент чипсеты практически полностью совместимы "снизу вверх". Но есть и исключение. Независимыми разработчиками представлено несколько видеокарт, являющихся как бы "добавочным" видеоконтроллером для Амиги. На Амигу можно поставить одну или несколько таких карт. Причиной для их разработки явилось некоторое отставание возможностей чипсета от требований рынка профессиональных рабочих станций, а в настоящее время — даже и от требований к чисто домашним машинам. Видеокарты предоставляют возможность использования больших, нежели чипсет, графических разрешений, и более сложных видеорежимов с большим количеством цветов. Поддержка видеокарт — это несколько отдельных библиотек, подменяющих стандартные graphics, intuition и т.д., и поставляющиеся в составе пакета CyberGraphX, прилагаемого к большинству карт. Кроме реализации всех стандартных графических возможностей системы, этот пакет предоставляет новые возможности — работа с 15, 16, 24 и 32-битным цветом, поддержка новых видеорежимов, поддержка нескольких одновременно доступных видеоконтроллеров и т.п. В настоящее время этот пакет стал стандартом де-факто. Любая программа, использующая только стандартные возможности системы, будет работать и на видеокарте, но здесь есть некоторые тонкости. Структура видеопамати видеокарт обычно совершенно не похожа на структуру видеопамати чипсета: например, 24-битный цвет реализован не в виде 24 битпланов, а в виде трех или четырех байт на точку. Видеокарты не имеют копера, и по этой причине несколько открытых на карте экранов не имеют отдельной видеопамати. Управляющие регистры карт совсем не похожи на регистры чипсета. Карты не имеют режимов HAM, ENB, DPF... Кроме того, карты по регистрам и возможностям несовместимы друг с другом.

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

Описание отличий можно вести еще долго, но из этого следует вывести более практичный вывод: программируйте корректно через систему! Это единственный способ достичь работоспособности Вашей программы на любых конфигурациях, нынешних и будущих...

Еще одно замечание о графике касается следующего. Исторически сложилось так, что AMIGA — машина, по своей сути, с телевизионной разверткой. У большинства пользователей стоят телевизоры и телевизионные мониторы, и обычно предполагается, что монитор, стоящий на Амиге, “тянет” телевизионную развертку. Из-за этого получается, что многие программы, в лучшем случае, открывают экраны в телевизионном режиме, а в худшем — путем программирования чипсета напрямую работают с графикой сами. А мониторы могут быть разные. В последнее время можно достаточно дешево купить VGA-монитор от IBM PC, который при умеренной стоимости (особенно, если он монохромный) обеспечит весьма и весьма приличное качество “картинки”, но при этом неспособен работать в телевизионном стандарте.

Конечно же, гораздо лучше с практической точки зрения купить Multisync монитор, который “потянет” любые частоты, но не следует забывать и о менее обеспеченных пользователях, желающих видеть на экране качественную картинку без “интерлесинга”. Для них работа с такими программами может превратиться в сущий ад. Несколько известных программ (самые яркие примеры — SoundTracker Pro, ProTracker, DigiBooster) работают с чипсетом “сами”, в результате чего их работа на VGA-мониторе невозможна в принципе. Из-за этого пользователи вынуждены держать “под рукой” телевизор или телевизионный монитор. Выход из такой ситуации прост — опять же, пользуйтесь операционной системой! Она предоставляет Вам возможность открыть экран в любом разрешении и режиме. Впрочем, даже если Ваша программа и не позволит изменить режим открываемого экрана, есть множество пакетов (например,

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

ModePro, NewMode, MCP и т.п.), позволяющих отслеживать открытие экранов любыми процессами и преобразовывать запрашиваемые программами видеорежимы в такие, которые больше нравятся пользователю, а также переадресовывать открытие экранов чипсета на видеокарты.

И последний момент: не увлекайтесь использованием спрайтов! На карточках их нет, да и во многих режимах чипсета, отличающихся от PAL и NTSC, все восемь спрайтов, в силу некоторых особенностей, просто недоступны, и гарантируется наличие только одного спрайта — мыши.

Несколько слов о многозадачности.

Теперь о “многозадачности”, точнее о ее конкретной реализации в AmigaOS. Каждый процесс имеет свой приоритет, а при постановке процессов в очередь на исполнение, эта очередь сортируется по приоритетам процессов: имеющий наибольший приоритет процесс получит управление первым. Процессы, имеющие одинаковые приоритеты, будут выполняться одновременно.

Пользовательские задачи, работающие на нулевом приоритете, будут выполняться параллельно. Тут кроется основная ошибка начинающих программистов — из того, что задачи будут работать одновременно, делается неправильный вывод: при ожидании чего-либо задача может просто крутиться в цикле, например, проверяя состояние какого-то бита и постоянно переходя на эту проверку, если проверка показала, что состояние не изменилось.

Такое еще можно простить программам в однозадачных системах. На Амиге же это приводит к тому, что эта задача начинает “зажирать” некоторый процент машинного времени, заметно мешая остальным задачам (для пользователя это выглядит так, как будто остальные задачи начинают “тормозить”). Кроме того, задачи с меньшим приоритетом рискуют вообще не получить управления, и будут стоять на месте до завершения такого процесса. Такой подход в корне неверен, и задаче, ожидающей какого-либо события, следует воспользоваться средствами “отда-

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

чи" управления системе "до лучших времен".

Другая тонкость многозадачности состоит в следующем: принятая в AmigaOS схема распределения времени приводит к тому, что при работе какого-либо процесса, не отдающего никому управление, процесс с меньшим приоритетом никогда не получит управления, так как будет всегда становиться в очередь на исполнение за ним. Этой особенностью системы иногда пользуются некоторые программы, используя ее для внутреннего планирования времени процессора. Однако при установке известного пакета Executive, который с целью улучшения работы планировщика разделения времени начинает манипулировать приоритетами процессов, вполне может наступить такая ситуация, что приоритеты этих двух процессов "поменяются местами", и работоспособность программы, основанной на этой особенности системы, нарушится. Executive имеет средства выключения заданных процессов из списка динамических приоритетов, но лучшим подходом все-же следует считать просто отказ от применения подобных трюков.

"Что делать?"

Теперь, практически, самое главное. Вы вполне серьезно решили писать что-то свое. Следует сразу определиться: как Вы собираетесь это писать, и что это будет... Давно известно, что программы на Амиге делятся на две большие группы: нечто красивое и, как правило, бесполезное практически (demo и прочее) и функционально завершенные прикладные и системные программы, ориентированные на работу под операционкой и использующие, в основном, ее возможности.

Как Вы, наверное, уже заметили, наши симпатии, в основном, на стороне "дружественных к системе" ("system-friendly") программ. С другой стороны, никто не мешает "скрестить коня и трепетную лань" — писать программы так, чтобы они предельно корректно работали под ОС и использовали возможности машины "на всю катушку"... Рассмотрим по пунктам, что же для этого нужно:

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ?

1) По возможности избегайте прямого программирования "железа". Там, где его можно избежать полностью, лучше так и сделать; там же, где использование железа напрямую резко увеличивает производительность работы — лучше сделать выбор. Например, как это сделано в известной программе CygnusED: скроллинг экрана можно делать средствами программы или средствами системы. Первый вариант быстрее (быстрее он, правда, только на чипсете), но только второй вариант делает возможной эксплуатацию этой программы на видеокартах и открытие окошек на ее экране.

2) По возможности используйте, упомянутую в книге возможность прямого обращения к битпланам открытого экрана. Например, на видеокартах это попросту не будет работать, а на экранах чипсета не даст возможности открывать окна другим процессам. Имеет смысл поступить, как описано выше — если уж скорость "прижимает", сделайте опцию типа "Custom routines", как в известной программе TextView.

3) Работайте с диском ТОЛЬКО через вызовы dos.library. Никаких trackloader-ов и прочих извращений! Конечно, это довольно красиво — одновременно идущая подгрузка с дискеты, музыка и зфффекты на экране в демке, но программы, которые работают исключительно с дискет, могут испортить нервы кому угодно. Известны и про более "тяжелые случаи" — программы, которые для работы с винчестером самостоятельно сканировали файловую систему и определяли местонахождение отдельных секторов, составляющих файлы, после чего залезали на низком уровне в порты IDE винчестера и считывали информацию самостоятельно. Естественно, такие программы не могут работать ни на SCSI винчестерах, ни на любых других файловых системах...

4) Ни в коем случае не вызывать процедуры из ПЗУ системы по абсолютным адресам (у разных версий ПЗУ они разные!). Не пытаться самостоятельно предполагать, где

AMIGA СЕГОДНЯ: ЧТО ИЗМЕНИЛОСЬ

может находиться fast-память, и самостоятельно с ней работать, минуя системный механизм распределения памяти.

5) Не использовать недокументированные возможности системы. Никогда не заполняйте зарезервированные биты, никогда не вызывайте зарезервированные точки входа библиотек. Впрочем, то же самое касается и “железа” — довольно большой процент неработающих на AGA OCS'ных программ не работает по причине использования резервных полей и битов управляющих регистров.

Какую цель преследует все вышеперечисленное? Цель очень проста — гарантировать работоспособность программы на всех возможных и достаточных для работы программы конфигурациях, и работу без особых проблем параллельно с любыми другими задачами. Конечно же, все вышесказанное не претендует на абсолютную истину, и никто Вам не мешает делать программы так, как Вам заблагорассудится. Более того, если программы делаются чисто “для служебного пользования”, особого значения это тоже не имеет. Но не удивляйтесь, что Ваша программа вдруг перестанет работать после очередного “апгрейда” или же после установки какого-нибудь другого программного пакета.

ПРИЛОЖЕНИЯ**ПРИЛОЖЕНИЯ****1. Обзор библиотечных функций AMIGA OS 1.3**

Эта часть приложения содержит список основных функций библиотек операционной системы. Для каждой функции приводится смещение (в десятичном и шестнадцатеричном форматах), название и набор параметров. Например, строка

-**\$0228 -552** **OpenLibrary(libName,version)** (A1,D0) описывает функцию **OpenLibrary** (открыть библиотеку), параметрами которой являются имя библиотеки (**libName**) и номер версии (**version**). Эти параметры передаются в регистрах A1 и D0 соответственно.

CLIST.LIBRARY Для работы с коррект-списком

\$001E -30 **InitCLPool (cLPool,size)** (A0,D0)
\$0024 -36 **AllocCList (cLPool)** (A1)
\$002A -42 **FreeCList (cList)** (A0)
\$0030 -48 **FlushCList (cList)** (A0)
\$0036 -54 **SizeCList (cList)** (A0)
\$003C -60 **PutCLChar (cList,byte)** (A0,D0)
\$0042 -66 **GetCLChar (cList)** (A0)
\$0048 -72 **UnGetCLChar (cList,byte)** (A0,D0)
\$004E -78 **UpPutCLChar (cList)** (A0)
\$0054 -84 **PutCLWord (cList,word)** (A0,D0)
\$005A -90 **GetCLWord (cList)** (A0)
\$0060 -96 **UnGetCLWord (cList,word)** (A0,D0)
\$0066 -102 **UnPutCLWord (cList)** (A0)
\$006C -108 **PutCLBuf (cList,buffer,length)** (A0,A1,D1)
\$0072 -114 **GetCLBuf (cList,buffer,maxLength)**
(A0,A1,D1)

ПРИЛОЖЕНИЯ

- \$0078 -120 MarkCList (cList,offset) (A0,D0)
- \$007E -126 IncrCLMark (cList) (A0)
- \$0084 -132 PeekCLMark (cList) (A0)
- \$008A -138 SplitCList (cList) (A0)
- \$0090 -144 CopyCList (cList) (A0)
- \$0096 -150 SubCList (cList,index,length) (A0,D0,D1)
- \$009C -156 ConcatCList (sourceCList,destCList) (A0,A1)

CONSOLE.LIBRARY I/O глa работи с консолю

- \$002A -42 CDInputHandler (events,device) (A0,A1)
- \$0030 -48 RawKeyConvert (events,buffer,length,keyMap)
(A0,A1,D1,A2)

DISKFONT.LIBRARY глa работи со шрифтови, записани на диск

- \$001E -30 OpenDiskFont (textAttr) (A0)
- \$0024 -36 AvailFonts (buffer,bufBytes,flags) (A0,D0,D1)

DOS.LIBRARY I/O на екран, файл и т.п.

- \$001e -30 Open (name,access mode)(d1,d2)
- \$0024 -36 Close (file)(d1)
- \$002a -42 Read (file,buffer,length)(d1,d2,d3)
- \$0030 -48 Write (file,buffer,length)(d1,d2,d3)
- \$0036 -54 Input()
- \$003c -60 Output()
- \$0042 -66 Seek(file,position,offset)(d1,d2,d3)
- \$0048 -72 DeleteFile (name)(d1)
- \$004e -78 Rename(oldname,newname)(d1,d2)
- \$0054 -84 Lock(name,type)(d1,d2)
- \$005a -90 UnLock(lock)(d1)
- \$0060 -96 DupLock(lock)(d1)
- \$0066 -102 Examine(lock,fileinfofblock)(d1,d2)
- \$006c -108 ExNext(lock,fileinfofblock)(d1,d2)
- \$0072 -114 Info(lock,parameterblock)(d1,d2)
- \$0078 -120 CreateDir(name)(d1)
- \$007e -126 CurrentDir(lock)(d1)
- \$0084 -132 IoErr()
- \$008a -138 CreateProc(name,pri,seglist,stacksize)
(d1,d2,d3,d4)
- \$0090 -144 Exit(returncode)(d1)

ПРИЛОЖЕНИЯ

\$0096	-150	LoadSeg(filename)(d1)
\$009c	-156	UnLoadSeg(segment)(d1)
\$00a2	-162	Getpacket(wait)(d1)
\$00a8	-168	Queuepacket(packet)(d1)
\$00ae	-174	DeviceProc(name)(d1)
\$00be	-180	SetComment(name,comment)(d1,d2)
\$00ba	-186	SetProtection(name,mask)(d1,d2)
\$00c0	-192	DateStamp(date)(d1)
\$00c6	-198	Delay(timeout)(d1)
\$00cc	-204	WaitForChar(file,timeout)(d1,d2)
\$00d2	-210	ParentDir(lock)(d1)
\$00d8	-216	IsInteractive(file)(d1)
\$00de	-222	Execute(string,file,file)(d1,d2,d3)

EXEC.LIBRARY *Такоже функції (находяться в ROM)*

\$001e	-30	Supervisor()
\$0024	-36	ExitIntr()
\$002a	-42	Schedule()
\$0030	-48	Reschedule()
\$0036	-54	Switch()
\$003c	-60	Dispatch()
\$0042	-66	Exception()
\$0048	-72	InitCode(startclass,version)(d0,d1)
\$004e	-78	InitStruct(inittable,memory,size)(a1,a2,d0)
\$0054	-84	MakeLibrary(funcinit,structinit,libinit, datasize,code size)(a0,a1,a2,d0,d1)
\$005a	-90	MakeFunctions(target,functionarray, funcdispbase)(a0,a1,a2)
\$0060	-96	FindResident(name)(a1)
\$0066	-102	InitResident(resident,seglist)(a1,d1)
\$006c	-108	Alert(alertnum,parameters)(d7,a5)
\$0072	-114	Debug()
\$0078	-120	Disable()
\$007e	-126	Enable()
\$0084	-132	Forbid()
\$008a	-138	Permit()
\$0090	-144	SetSR(newsr,mask)(d0,d1)

ПРИЛОЖЕННЯ

\$0096	-150	SuperState()
\$009c	-156	UserState(sysstack)(d0)
\$00a2	-162	setIntVector(intnumber,interrupt)(d0,d1)
\$00a8	-168	AddIntServer(intnumber,interrupt)(d0,d1)
\$00ae	-174	RemIntServer(intnumber,interrupt)(d0,d1)
\$00b4	-180	Cause(interrupt)(a1)
\$00ba	-186	Allocate(freelist,bytesize)(a0,d0)
\$00c0	-192	Deallocate(freelist,memoryblock,bytesize)(a0,a1,d0)
\$00c6	-198	AllocMem(bytesize,requirements)(d0,d1)
\$00cc	-204	AlloAbs(bytesize,location)(d0,a1)
\$00d2	-210	FreeMem(memoryblock,bytesize)(a1,d0)
\$00d8	-216	AvailMem(requirements)(d1)
\$00de	-222	AllocEntry(entry)(a0)
\$00e4	-228	FreeEntry(entry)(a0)
\$00ea	-234	Insert(list,node,pred)(a0,a1,a2)
\$00f0	-240	AddHead(list,node)(a0,a1)
\$00f6	-246	AddTail(list,node)(a0,a1)
\$00fc	-252	Remove(node)(a1)
\$0102	-258	RemHead(list)(a0)
\$0108	-264	RemTail(list)(a0)
\$010e	-270	Enqueue(list,node)(a0,a1)
\$0114	-276	FindName(list,name)(a0,a1)
\$011a	-282	AddTask(task,initpc,finalpc)(a1,a2,a3)
\$0120	-288	RemTask(task)(a1)
\$0126	-294	FindTask(name)(a1)
\$012c	-300	SetTaskPri(task,priority)(a1,d0)
\$0132	-306	SetSignal(newsignals,signalset)(d0,d1)
\$0138	-312	SetExcept(newsignals,signalset)(d0,d1)
\$013e	-318	Wait(signalset)(d0)
\$0144	-324	Signal(task,signalset)(a1,d0)
\$014a	-330	AllocSignal(signalnum)(d0)
\$0150	-336	FreeSignal(signalnum)(d0)
\$0156	-342	AllocTrap(trapnum)(d0)
\$015c	-348	FreeTrap(trapnum)(d0)
\$0162	-354	AddPort(port)(a1)

ПРИЛОЖЕНИЯ

\$0168	-360	RemPort(port)(a1)
\$016e	-366	PutMsg(port,message)(a0,a1)
\$0174	-372	GetMsg(port)(a0)
\$017a	-378	ReplyMsg(message)(a1)
\$0180	-384	WaitPort(port)(a0)
\$0186	-390	FindPort(name)(a1)
\$018c	-396	AddLibrary(library)(a1)
\$0192	-402	RemLibrary(library)(a1)
\$0198	-408	OldOpenLibrary(libname)(a1)
\$019e	-414	CloseLibrary(library)(a1)
\$01a4	-420	Setfunction(library,funcoffset,funcentry) (a1,a0,d0)
\$01aa	-426	SumLibrary(library)(a1)
\$01b0	-432	AddDevice(device)(a1)
\$01b6	-438	RemDevice(device)(a1)
\$01bc	-444	OpenDevice(devname,unit,iorequest,flags) (a0,d0,a1,d1)
\$01c2	-450	CloseDevice(iorequest)(a1)
\$01c8	-456	DoIO(iorequest)(a1)
\$01ce	-462	SendIO(iorequest)(a1)
\$01d4	-468	CheckIO(iorequest)(a1)
\$01da	-474	WaitIO(iorequest)(a1)
\$01e0	-480	AbortIO(iorequest)(a1)
\$01e6	-486	AddResource(resource)(a1)
\$01ec	-492	RemResource(resource)(a1)
\$01f2	-498	OpenResource(resname,version)(a1,d0)
\$01f8	-504	RawIOInit()
\$01fe	-510	RawMayGetChar()
\$0204	-516	RawPutChar(char)(d0)
\$020a	-522	RawDoFmt()(a0,a1,a2,a3)
\$0210	-528	GetCC()
\$0216	-534	TypeOfMem(address)(a1)
\$021c	-540	Procedure(semaphore,bidmsg)(a0,a1)
\$0222	-546	Vacate(semaphore)(a0)
\$0228	-552	OpenLibrary(libname,version)(a1,d0)

ПРИЛОЖЕНИЯ

GRAPHICS.LIBRARY для работы с blitter-сопроцессором и графикой

\$001e	-30	BltBitMap(srcbitmap,scrx,scry,destbitmap,destx,desty,sizex,sizey,minterm,mask,tempa)(a0,d0,d1,a1,d2,d3,d4,d5,d6,d7,a2)
\$0024	-36	BltTemplate(source,scrx,scrmod,destrastport,destx,desty,sizex,sizey)(a0,d0,d1,a1,d2,d3,d4,d5)
\$002a	-42	ClearEOL(rastport)(a1)
\$0030	-48	ClearScreen(rastport)(a1)
\$0036	-54	TextLength(rastport,string,count)(a1,a0,d0)
\$003c	-60	Text(rastport,string,count)(a1,a0,d0)
\$0042	-66	SetFont(rastportid,textfont)(a1,a0)
\$0048	-72	CpenFont(textattr)(a0)
\$004e	-78	CloseFont(textfont)(a1)
\$0054	-84	AskSoftStyle(rastport)(a1)
\$005a	-90	SetSoftStyle(rastport,style,enable)(a1,d0,d1)
\$0060	-96	AddBob(bob,rastport)(a0,a1)
\$0066	-102	AddVSprite(vsprite,rastport)(a0,a1)
\$006c	-108	DoCollision(rastport)(a1)
\$0072	-114	DrawGLList(rastport,viewport)(a1,a0)
\$0078	-120	InitGels(dummyhead,dummytail,gelsinfo)(a0,a1,a2)
\$007e	-126	InitMasks(vsprite)(a0)
\$0084	-132	RemIBob(bob,rastport,viewport)(a0,a1,a2)
\$008a	-138	RemVSprite(vsprite)(a0)
\$0090	-144	SetCollision(type,routine,gelsinfo)(d0,a0,a1)
\$0096	-150	SortGLList(rastport)(a1)
\$009c	-156	AddAnimObj(obj,animationkey,rastport)(a0,a1,a2)
\$00a2	-162	Animate(animationkey,rastport)(a0,a1)
\$00a8	-168	GetBuffers(animationobj,rastport,doublebuffer)(a0,a1,d0)
\$00ae	-174	InitGMasks(animationobj)(a0)
\$00b4	-180	GelsFuncE()
\$00ba	-186	GelsFuncF()
\$00c0	-192	LoadRGB4(viewport,colours,count)(a0,a1,d0)

ПРИЛОЖЕНИЯ

\$00c6	-198	InitRastPort(rastport)(a1)
\$00cc	-204	InitVPort(viewport)(a0)
\$00d2	-210	MrgCop(view)(a1)
\$00D8	-216	MakeVPort (view,viewPort) (A0,A1)
\$00DE	-222	LoadView (view) (A1)
\$00E4	-228	WaitBlit ()
\$00EA	-234	SetRast (rastPort,color) (A1,D0)
\$00F0	-240	Move (rastPort,x,y) (A1,D0,D1)
\$00F6	-246	Draw (rastPort,x,y) (A1,D0,D1)
\$00FC	-252	AreaMove (rastPort,x,y) (A1,D0,D1)
\$0102	-258	AreaDraw (rastPort,x,y) (A1,D0,D1)
\$0108	-264	AreaEnd (rastPort) (A1)
\$010E	-270	WaitTOF ()
\$0114	-276	QBlit (blit) (A1)
\$011A	-282	InitArea (areaInfo,vectorTable, vectorTableSize)(A0,A1,D0)
\$0120	-288	SetRGB4 (viewPort,index,r,g,b) (A0,D0,D1,D2,D3)
\$0126	-294	QBSBlit (blit) (A1)
\$012C	-300	BltClear (memory,size,flags) (A1,D0,D1)
\$0132	-306	RectFill (rastPort,xl,yl,xu,yu) (A1,D0,D1,D2,D3)
\$0138	-312	BltPattern (rastPort,ras,xl,yl,maxX,maxY, fillBytes) (A1,A0,D0,D1,D2,D3,D4)
\$013E	-318	ReadPixel (rastPort,x,y) (A1,D0,D1)
\$0144	-324	WritePixel (rastPort,x,y) (A1,D0,D1)
\$014A	-330	Flood (rastPort,mode,x,y) (A1,D2,D0,D1)
\$0150	-336	PolyDraw (rastPort,count,polyTable) (A1,D0,A0)
\$0156	-342	SetAPen (rastPort,pen) (A1,D0)
\$015C	-348	SetBPen (rastPort,pen) (A1,D0)
\$0162	-354	SetDrMd (rastPort,drawMode) (A1,D0)
\$0168	-360	InitView (view) (A1)
\$016E	-366	CBump (copperList) (A1)
\$0174	-372	CMove (copperList,destination,data) (A1,D0,D1)

ПРИЛОЖЕНИЯ

\$017A -378	CWait (copperList,x,y) (A1,D0,D10)
\$0180 -384	VBeamPos ()
\$0186 -390	InitBitMap (bitMap,depth,width,height) (A0,D0,D1,D2)
\$018C -396	ScrollRaster (rastPort,dX,dY,minx,miny,maxx, maxy) (A1,D0,D1,D2,D3,D4,D5)
\$0192 -402	WaitBOVP (viewPort) (A0)
\$0198 -408	GetSprite (simpleSprite,num) (A0,D0)
\$019E -414	FreeSprite (num) (D0)
\$01A4 -420	ChangeSprite (vp.simpleSprite,data) (A0,A1,A2)
\$01AA -426	MoveSprite (viewPort.simpleSprite,x,y) (A0,A1,D0,D1)
\$01B0 -432	LockLayerRom (layer) (A5)
\$01B6 -438	UnlockLayerRom (layer) (A5)
\$01BC -444	SyncSBitMap (1) (A0)
\$01C2 -450	CopySBitMap (11,12) (A0,A1)
\$01C8 -456	OwnBlitter ()
\$01CE -462	DisownBlitter ()
\$01D4 -468	InitTmpRas (tmpras,buff,size) (A0,A1,D0)
\$01DA -474	AskFont (rastPort,textAttr) (A1,A0)
\$01E0 -480	AddFont (textFont) (A1)
\$01E6 -486	RemFont (textFont) (A1)
\$01EC -492	AllocRaster (width,height) (D0,D1)
\$01F2 -498	FreeRaster (planePtr,width,height) (A0,D0,D1)
\$01F8 -504	AndRectRegion (rgn,rect) (A0,A1)
\$01FE -510	OrRectRegion (rgn,rect) (A0,A1)
\$0204 -516	NewRegion ()
\$020A -522	** зарезервировано **
\$0210 -528	ClearRegion (rgn) (A0)
\$0216 -534	DisposeRegion (rgn) (A0)
\$021C -540	FreeVPortCopLists (viewPort) (A0)
\$0222 -546	FreeCopList (coplist) (A0)
\$0228 -552	ClipBlit(srcrp,srcX,srcY,destrp,destX,destY, sizeX,sizeY,minterm)(A0,D0,D1,A1,D2,D3,D4, D5,D6)

ПРИЛОЖЕНИЯ

\$022E	-558	XorRectRegion (rgn,rect) (A0,A1)
\$0234	-564	FreeCprList (cprlist) (A0)
\$023A	-570	GetColorMap (entries) (D0)
\$0240	-576	FreeColorMap (colormap) (A0)
\$0246	-582	GetRGB4 (colormap,entry) (A0,D0)
\$024C	-588	ScrollVPort (vp) (A0)
\$0252	-594	UCopperListInit (copperlist,num) (A0,D0)
\$0258	-600	FreeGBuffers (animationObj,rastPort, doubleBuffer) (A0,A1,D0)
\$025E	-606	BltBitMapRastPort(srcbm,srcx,srcy,dest rp, destX,destY,sizeX,sizeY,minter)(A0,D0,D1,A1, D2,D3,D4,D5,D6)

ICON.LIBRARY *для работы с иконками Workbench*

\$001E	-30	GetWBOobject (name) (A0)
\$0024	-36	PutWBOobject (name,object) (A0,A1)
\$002A	-42	GetIcon (name,icon,freelist) (A0,A1,A2)
\$0030	-48	PutIcon (name,icon) (A0,A1)
\$0036	-54	FreeFreeList (freelist) (A0)
\$003C	-60	FreeWBOobject (WBOobject) (A0)
\$0042	-66	AllocWBOobject ()
\$0048	-72	AddFreeList (freelist,mem,size) (A0,A1,A2)
\$004E	-78	GetDiskObject (name) (A0)
\$0054	-84	PutDiskObject (name,diskobj) (A0,A1)
\$005A	-90	FreeDiskObj (diskobj) (A0)
\$0060	-96	FindToolType (toolTypeArray,typeName) (A0,A1)
\$0066	-102	MatchToolValue (typeString,value) (A0,A1)
\$006C	-108	BumbRevision (newname,oldname) (A0,A1)

INTUITION.LIBRARY *для работы с GUI*

\$001E	-30	OpenIntuition ()
\$0024	-36	Intuition (ievent) (A0)
\$002A	-42	AddGadget (AddPtr,Gadget,Position) (A0,A1,D0)
\$0030	-48	ClearDMRequest (Window) (A0)
\$0036	-54	ClearMenuStrip (Window) (A0)
\$003C	-60	ClearPointer (Window) (A0)

ПРИЛОЖЕНИЯ

\$0042	-66	CloseScreen (Screen) (A0)
\$0048	-72	CloseWindow (Window) (A0)
\$004E	-78	CloseWorkbench ()
\$0054	-84	CurrentTime (Seconds, Micros) (A0, A1)
\$005A	-90	DisplayAlert (AlertNumber, String, Height) (D0, A0, D1)
\$0060	-96	DisplayBeep (Screen) (A0)
\$0066	-102	DoubleClick (seconds, micros, cseconds, cmicros) (D0, D1, D2, D3)
\$006C	-108	DrawBorder (Rport, Border, LeftOffset, TopOffset) (A0, A1, D0, D1)
\$0072	-114	DrawImage (Rport, Image, LeftOffset, TopOffset) (A0, A1, D0, D1)
\$0078	-120	EndRequest (requester, window) (A0, A1)
\$007E	-126	GetDefPref (preferences, size) (A0, D0)
\$0084	-132	GetPrefs (preferences, size) (A0, D0)
\$008A	-138	InitRequester (req) (A0)
\$0090	-144	ItemAddress (MenuStrip, MenuNumber) (A0, D0)
\$0096	-150	ModifyIDCMP (Window, Flags) (A0, D0)
\$009C	-156	ModifyProp (Gadget, Ptr, Reg, Flags, HPos, VPos, HBody, VBody) (A0, A1, A2, D0, D1, D2, D3, D4)
\$00A2	-162	MoveScreen (Screen, dx, dy) (A0, D0, D1)
\$00A8	-168	MoveWindow (Window, dx, dy) (A0, D0, D1)
\$00AE	-174	OffGadget (Gadget, Ptr, Req) (A0, A1, A2)
\$00B4	-180	OffMenu (Window, MenuNumber) (A0, D0)
\$00BA	-186	OnGadget (Gadget, Ptr, Req) (A0, A1, A2)
\$00C0	-192	OnMenu (Window, MenuNumber) (A0, D0)
\$00C6	-198	OpenScreen (OSArgs) (A0)
\$00CC	-204	OpenWindow (OWArgs) (A0)
\$00D2	-210	OpenWorkBench ()
\$00D8	-216	PrintIText (rp, itext, left, top) (A0, A1, D0, D1)
\$00DE	-222	RefreshGadgets (Gadgets, Ptr, Req) (A0, A1, A2)
\$00E4	-228	RemoveGadgets (RemPtr, Gadget) (A0, A1)
\$00EA	-234	ReportMouse (Window, Boolean) (A0, D0)
\$00F0	-240	Request (Requester, Window) (A0, A1)

ПРИЛОЖЕНИЯ

\$00F6	-246	ScreenToBack (Screen) (A0)
\$00FC	-252	ScreenToFront (Screen) (A0)
\$0102	-258	SetDMRequest (Window,req) (A0,A1)
\$0108	-264	SetMenuStrip (Window,Menu) (A0,A1)
\$010E	-270	SetPointer (Window,Pointer,Height,Width, XOFFset, YOFFset) (A0,A1,D0,D1,D2,D3)
\$0114	-276	SetWindowTitles (Window>windowTitle, screenTitle) (A0,A1,A2)
\$011A	-282	ShowTitle (Screen>ShowIt) (A0,D0)
\$0120	-288	SizeWindow (Window>mdx,dy) (A0,D0,D1)
\$0126	-294	ViewAddress ()
\$012C	-300	ViewPortAddress (Window) (A0)
\$0132	-306	WindowToBack (Window) (A0)
\$0138	-312	WindowToFront (Window) (A0)
\$013E	-318	WindowLimits (Window,minwidth,minheight, maxwidth,maxheight) (A0,D0,D1,D2,D3)
\$0144	-324	SetPrefs (preferences,size,flag) (A0,D0,D1)
\$014A	-330	IntuiTextLength (itext) (A0)
\$0150	-336	WBenchToBack ()
\$0156	-342	WBenchToFront ()
\$015C	-348	AutoRequest (Window,Body,PText,NText, PFlag,NFlag,W,H) (A0,A1,A2,A3,D0,D1,D2,D3)
\$0162	-354	BeginRefresh (Window) (A0)
\$0168	-360	BuildSysRequest(Window,Body,PosText, NegText,Flags,W,H) (A0,A1,A2,A3,D0,D1,D2)
\$016E	-366	EndRefresh (Window,Complete) (A0,D0)
\$0174	-372	FreeSysRequest (Window) (A0)
\$017A	-378	MakeScreen (Screen) (A0)
\$0180	-384	RemakeDisplay ()
\$0186	-390	RethinkDisplay ()
\$018C	-396	AllocRemember (RememberKey,Size,Flags) (A0,D0,D1)
\$0192	-402	AlohaWorkBench (wbport) (A0)
\$0198	-408	FreeRemember (RememberKey,ReallyForgot) (A0,D0)
\$019E	-414	LockIBase (dontknow) (D0)

ПРИЛОЖЕНИЯ

- \$01A4 -420 UnlockIBase (IBLock) (A0)
LAYERS.LIBRARY *для работы с экранной памятью*
 \$001E -30 InitLayers (li) (A0)
 \$0024 -36 CreateUpfrontLayer (li,bm,x0,y0,x1,y1,flags, bm2) (A0,A1,D0,D1,D2,D3,D4,A2)
 \$002A -42 CreateBehindLayer (li,bm,x0,y0,x1,y1,flags, bm2) (A0,A1,D0,D1,D2,D3,D3,A2)
 \$0030 -48 UpfrontLayer (li,layer) (A0,A1)
 \$0036 -54 BehindLayer (li,layer) (A0,A1)
 \$003C -60 MoveLayer (li,layer,dx,dy) (A0,A1,D0,D1)
 \$0042 -66 SizeLayer (li,layer,dx,dy) (A0,A1,D0,D1)
 \$0048 -72 ScrollLayer (li,layer,dx,dy) (A0,A1,D0,D1)
 \$004E -78 BeginUpdate (layer) (A0)
 \$0054 -84 EndUpdate (layer) (A0)
 \$005A -90 DeleteLayer (li,layer) (A0,A1)
 \$0060 -96 LockLayer (li,layer) (A0,A1)
 \$0066 -102 UnlockLayer (li,layer) (A0,A1)
 \$006C -108 LockLayers (li) (A0)
 \$0072 -114 UnlockLayers (li) (A0)
 \$0078 -120 LockLayerInfo (li) (A0)
 \$007E -126 SwapBitRastPortClipRect (rp,cr) (A0,A1)
 \$0084 -132 WhichLayer (li,x,y) (A0,D0,D1)
 \$008A -138 UnlockLayerInfo (li) (A0)
 \$0090 -144 NewLayerInfo ()
 \$0096 -150 DisposeLayerInfo (li) (A0)
 \$009C -156 FattenLayerInfo (li) (A0)
 \$00A2 -162 ThinLayerInfo (li) (A0)
 \$00A8 -168 MoveLayerInfrontOf (layer_to_move, layer_to_be_in_front_of) (A0,A1)
MATHFFP.LIBRARY *основные мат. операции с плавающей точкой*
 \$001E -30 SPFix (float) (D0)
 \$0024 -36 SPFit (integer) (D0)
 \$002A -42 SPCmp (leftFloat,right,Float) (D1,D0)
 \$0030 -48 SPTst (float) (D1)
 \$0036 -54 SPAbs (float) (D0)
 \$003C -60 SFLNeg (float) (D0)

ПРИЛОЖЕНИЯ

\$0042	-66	SPAdd (leftFloat,rightFloat) (D1,D0)
\$0048	-72	SPSub (leftFloat,rightFloat) (D1,D0)
\$004E	-78	SPMul (leftFloat,rightFloat) (D1,D0)
\$0054	-84	SPDiv (leftFloat,rightFloat) (D1,D0)

MATHEEDOUBAS.LIBRARY *мат. функции для работы с целыми числами*

\$001E	-30	IEEEDPFix (integer,integer) (D0,D1)
\$0024	-36	IEEEDPFit (integer) (D0)
\$002A	-42	IEEEDPComp (integer,integer,integer, integer) (D0,D1,D2,D3)
\$0030	-48	IEEEDPtst (integer,integer) (D0,D1)
\$0036	-54	IEEEDPAbs (integer,integer) (D0,D1)
\$003C	-60	IEEEDPNeg (integer,integer) (D0,D1)
\$0042	-66	IEEEDPAdd (integer,integer,integer,integer) (D0,D1,D2,D3)
\$0048	-72	IEEEDPSub (integer,integer,integer,integer) (D0,D1,D2,D3)
\$004E	-78	IEEEDPMul (integer,integer,integer,integer) (D0,D1,D2,D3)
\$0054	-84	IEEEDPDiv (integer,integer,integer,integer) (D0,D1,D2,D3)

MATHTRANS.LIBRARY *содержит некоторые элементарные функции*

\$001E	-30	SPAtan (float) (D0)
\$0024	-36	SPSin (float) (D0)
\$002A	-42	SPCos (float) (D0)
\$0030	-48	SPTan (float) (D0)
\$0036	-54	SPSincos (leftFloat,rightFloat) (D1,D0)
\$003C	-60	SPSinh (float) (D0)
\$0042	-66	SPCosh (float) (D0)
\$0048	-72	SPTanh (float) (D0)
\$004E	-78	SPExp (float) (D0)
\$0054	-84	SPLog (float) (D0)
\$005A	-90	SPPow (leftFloat,rightFloat) (D1,D0)
\$0060	-96	SPSqrt (float) (D0)
\$0066	-102	SPTieee (float) (D0)
\$006C	-108	SPFieee (float) (D0)
\$0072	-114	SPAsin (float) (D0)

ПРИЛОЖЕНИЯ

\$0078 -120 SPACos (float) (D0)
 \$007E -126 SPLog10 (float) (D0)
POTGO.LIBRARY для работы с аналоговыми устройствами ввода
 \$0006 -6 AllocPotBits (bits) (D0)
 \$000C -12 FreePotBits (bits) (D0)
 \$0012 -18 WritePotgo (word,mask) (D0,D1)
TIMER.LIBRARY для работы с таймерами
 \$002A -42 AddTime (dest,src) (A0,A1)
 \$0030 -48 SubTime (dest,src) (A0,A1)
 \$0036 -54 CmpTime (dest,src) (A0,A1)
TRANSLATOR.LIBRARY для перевода текста в форматированное представление
 \$001E -30 Translate (inputString,inputLength,
 outputBuffer,bufferSize) (A0,D0,A1,D1)

2. Обзор команд процессора MC68000

Используемые сокращения:

Label	метка (адрес)
Reg	регистр
Regs	список регистров
An	адресный регистр
Dn	регистр данных
Source	операнд источника
Dest	операнд приемника
<ea>	адрес или регистр
#0	абсолютное значение

Формат	Действия
ABCD Source, Dest	сложение двух BCD чисел
ADD Source, Dest	сложение
ADDA Source, An	сложение с адресным регистром
ADDI #n, <ea>	сложение с константой
ADDQ #n, <ea>	быстрое сложение с трехбитной константой
ADDX Source, Dest	сложение с переносом
AND Source, Dest	логическое И
ANDI #n, <ea>	логическое И с константой
ASL n, <ea>	арифметический сдвиг влево

ПРИЛОЖЕНИЯ

ASR	n,<ea>	арифметический сдвиг вправо
Bcc	Label	условный переход
BCHG	#n,<ea>	изменение бита
BCLR	#n,<ea>	очистка бита
BRA	Label	безусловный переход
BSET	#n,<ea>	установка бита
BSR	Label	ветвление к подпрограмме
BTST	#n,<ea>	проверка бита
CHK	<ea>,Dn	проверка регистра
CLR	<ea>	очистка (обнуление)
CMP	Source, Dest	сравнение
CMPA	<ea>,An	сравнение с адресным регистром
CMPI	#n,<ea>	сравнение с константой
CMPPM	Source, Dest	сравнение операндов в памяти
DBcc	Dn, Label	цикл (проверка, декремент и ветвление)
DIVS	Source, Dest	знаковое деление
DIVU	Source, Dest	беззнаковое деление
EOR	Source, Dest	исключающее ИЛИ
EORI	#n,<ea>	исключающее ИЛИ с константой
EXG	Reg, Reg	обмен регистров
EXT	Dn	расширение знака
JMP	Label	безусловный переход
JSR	Label	обращение к подпрограмме
LEA	<ea>,An	загрузка адреса
LINK	An, #n	выделение стекового фрейма
LSL	n,<ea>	логический сдвиг влево
LSR	n,<ea>	логический сдвиг вправо
MOVE	Source, Dest	пересылка
MOVE	SR,<ea>	пересылка из регистра статуса
MOVE	<ea>,SR	пересылка в регистр статуса
MOVE	<ea>,CCR	загрузка флагов
MOVE	USP,<ea>	пересылка из пользовательского указателя стека
MOVE	<ea>,USP	загрузка пользовательского указателя стека

ПРИЛОЖЕНИЯ

MOVEA	<ea>,An	пересылка в адресный регистр
MOVEM	Regs,<ea>	пересылка набора регистров
MOVEM	<ea>,Regs	загрузка набора регистров
MOVEP	Source,Dest	пересылка данных на периферийное устройство
MOVEQ	#n,Dn	быстрая пересылка восьмибитных данных
MULS	Source,Dest	знаковое умножение
MULU	Source,Dest	беззнаковое умножение
NBCD	Source,Dest	изменение знака BCD числа
NEG	<ea>	изменение знака
NEGX	<ea>	изменение знака с переносом
NOP		нет операции
NOT	<ea>	логическое НЕ (инвертирование)
OR	Source,Dest	логическое ИЛИ
ORI	#n,<ea>	логическое ИЛИ с константой
PEA	<ea>	загрузка адреса в стек
RESET		сброс внешних устройств
ROL	n,<ea>	циклический сдвиг через перенос влево
ROR	n,<ea>	циклический сдвиг через перенос вправо
ROXL	n,<ea>	циклический сдвиг влево
ROXR	n,<ea>	циклический сдвиг вправо
RTE		возврат из обработчика исключений
RTR		возврат из прерывания
RTS		возврат из подпрограммы
SBCD	Source,Dest	вычитание BCD чисел
Scc	<ea>	расширение условия
STOP		останов
SUB	Source,Dest	вычитание
SUBA	<ea>,An	вычитание из регистра адреса
SUBI	#n,<ea>	вычитание константы
SUBQ	#n,<ea>	быстрое вычитание трехбитной константы
SUBX	Source,Dest	вычитание с переносом

ПРИЛОЖЕНИЯ

SWAP	Dn	обмен старшего и младшего слов регистра
TAS	<ea>	проверка и установка бита 7
TRAP	#n	возбуждение исключения TRAP
TRAPV		возбуждение исключения TRAPV
UNLK	An	свертка стекового фрейма

3. Обзор команд всего ряда 680х0.

Принятые сокращения:

x - наличие операции у данного процессора

~ - инструкция режима супервизора

2 - программно эмулируемые инструкции

процессоров 68040 и 68060

Операция	Размер	68000	68010	68020	68030	68040	68060	6888x
abcd	b	x	x	x	x	x	x	-
add	b,w,l	x	x	x	x	x	x	-
addq	b,w,l	x	x	x	x	x	x	-
adda	w,l	x	x	x	x	x	x	-
addi	b,w,l	x	x	x	x	x	x	-
addx	b,w,l	x	x	x	x	x	x	-
and	b,w,l	x	x	x	x	x	x	-
andi	b,w,l	x	x	x	x	x	x	-
asr	b,w,l	x	x	x	x	x	x	-
asl	b,w,l	x	x	x	x	x	x	-
bcc	b,w,l	x	x	x	x	x	x	-
bchg	b,l	x	x	x	x	x	x	-
bclr	b,l	x	x	x	x	x	x	-
bchg	unsized	-	-	x	x	x	x	-
bclr	unsized	-	-	x	x	x	x	-
bfbxt	unsized	-	-	x	x	x	x	-
bfffo	unsized	-	-	x	x	x	x	-
bfins	unsized	-	-	x	x	x	x	-
bfset	unsized	-	-	x	x	x	x	-
bftst	unsized	-	-	x	x	x	x	-
bkpt	unsized	-	x	x	x	x	x	-
bset	b,l	x	x	x	x	x	x	-
btst	b,l	x	x	x	x	x	x	-
		68000	68010	68020	68030	68040	68060	6888x

AMIGA: ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ.

		ПРИЛОЖЕНИЯ						
ОПЕРАЦИЯ	РАЗМЕР	68000	68010	68020	68030	68040	68060	6888x
callm	unsized	-	-	x	-	-	-	-
cas	b,w,l	x	x	x	x	x	x,2	-
cas2	b,w,l	x	x	x	x	x	x,2	-
chk	b,w,l	x	x	x	x	x	x	-
chk2	b,w,l	-	-	x	x	x	2	-
cinv~	unsized	-	-	-	-	x	x	-
clr	b,w,l	x	x	x	x	x	x	-
cmp	b,w,l	x	x	x	x	x	x	-
cmpa	w,l	x	x	x	x	x	x	-
cmpi	b,w,l	x	x	x	x	x	x	-
cmpm	b,w,l	x	x	x	x	x	x	-
cmp2	b,w,l	-	-	x	x	x	2	-
cpush~	unsized	-	-	-	-	x	x	-
dbcc	w	x	x	x	x	x	x	-
divs	w,l	x	x	x	x	x	x,2	-
divsl	l	-	-	x	x	x	x	-
divu	w,l	x	x	x	x	x	x,2	-
divul	l	-	-	x	x	x	x	-
eor	b,w,l	x	x	x	x	x	x	-
eorl	b,w,l	x	x	x	x	x	x	-
eorl/								
ccr	b	x	x	x	x	x	x	-
eorl/								
sr~	w	x	x	x	x	x	x	-
exg	l	x	x	x	x	x	x	-
ext	w,l	x	x	x	x	x	x	-
extb	l	-	-	x	x	x	x	-
fabs	-	-	-	-	-	x	x	x
fsabs	-	-	-	-	-	x	x	-
fdabs	-	-	-	-	-	x	x	-
facos	-	-	-	-	-	2	2	x
fadd	-	-	-	-	-	x	x	x
fsadd	-	-	-	-	-	x	x	-
fdadd	-	-	-	-	-	x	x	-
fasin	-	-	-	-	-	2	2	x
fatan	-	-	-	-	-	2	2	x
		68000	68010	68020	68030	68040	68060	6888x

AMIGA: ПРОГРАММНЫЕ НА АССЕМБЛЕРА.

Операция	ПРИЛОЖЕНИЯ						
	68000	68010	68020	68030	68040	68060	68830
fatanh	-	-	-	-	2	2	x
fbcc	-	-	-	-	x	x	x
fcmp	-	-	-	-	x	x	x
fcos	-	-	-	-	2	2	x
fcosh	-	-	-	-	2	2	x
fdbcc	-	-	-	-	x	2	x
fdiv	-	-	-	-	x	x	x
fsdiv	-	-	-	-	x	x	-
fddiv	-	-	-	-	x	x	-
fetox	-	-	-	-	2	2	x
fetoxml	-	-	-	-	2	2	x
fgetexp	-	-	-	-	2	2	x
fgetman	-	-	-	-	2	2	x
fint	-	-	-	-	2	x	x
fintrz	-	-	-	-	2	x	x
flogn	-	-	-	-	2	2	x
flognpl	-	-	-	-	2	2	x
flog2	-	-	-	-	2	2	x
flog10	-	-	-	-	2	2	x
fmod	-	-	-	-	2	2	x
fmove	-	-	-	-	x	x	x
fsmove	-	-	-	-	x	x	-
fdmove	-	-	-	-	x	x	-
fmovecr	-	-	-	-	2	2	x
fmoveim	-	-	-	-	x	x,2	x
fmul	-	-	-	-	x	x	x
fsmul	-	-	-	-	x	x	-
fdmul	-	-	-	-	x	x	-
fneg	-	-	-	-	x	x	x
fsneg	-	-	-	-	x	x	-
fdneg	-	-	-	-	x	x	-
fnop	-	-	-	-	x	x	x
frem	-	-	-	-	2	2	x
frestore~	-	-	-	-	x	x	x
fscc	-	-	-	-	2	2	x
fsub	-	-	-	-	x	x	x
	68000	68010	68020	68030	68040	68060	68830

AMIGA: ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ.

		ПРИЛОЖЕНИЯ						
Символическое название	Размер	68000	68010	68020	68030	68040	68060	68080
fssub		-	-	-	-	x	x	-
fdsub		-	-	-	-	x	x	-
fsave~		-	-	-	-	x	x	x
fscale		-	-	-	-	2	2	x
fsglmul		-	-	-	-	2	2	x
fsgldiv		-	-	-	-	2	2	x
fsin		-	-	-	-	2	2	x
fsinh		-	-	-	-	2	2	x
fsincos		-	-	-	-	2	2	x
fsqrt		-	-	-	-	x	x	x
fssqrt		-	-	-	-	x	x	-
fdsqrt		-	-	-	-	x	x	-
ftan		-	-	-	-	2	2	x
ftanh		-	-	-	-	2	2	x
ftentox		-	-	-	-	2	2	x
ftrap		-	-	-	-	x	2	x
ftst		-	-	-	-	x	x	x
ftwotox		-	-	-	-	2	2	x
illegal	unsized x	x	x	x	x	x	x	-
jmp	unsized x	x	x	x	x	x	x	-
jsr	unsized x	x	x	x	x	x	x	-
lea	l	x	x	x	x	x	x	-
link	w,l	x	x	x	x	x	x	-
lpstop		-	-	-	-	-	x	-
lsl	b,w,l	x	x	x	x	x	x	-
lsr	b,w,l	x	x	x	x	x	x	-
move	b,w,l	x	x	x	x	x	x	-
movea	w,l	x	x	x	x	x	x	-
moveq	l	x	x	x	x	x	x	-
movec~l		-	x	x	x	x	x	-
movem	w,l	x	x	x	x	x	x	-
movep	w,l	x	x	x	x	x	-	-
moves~b,w,l		-	x	x	x	x	x	-
move16		-	-	-	-	x	x	-
muls	w,l	x	x	x	x	x	x	-
mulu	w,l	x	x	x	x	x	x	-

AMIGA: ПРОГРАММИРОВАНИЕ НА АССЕМБЛЕРЕ.

		ПРИЛОЖЕНИЯ						
Операция	Размер	68000	68010	68020	68030	68040	68060	6888x
nbcd	b	x	x	x	x	x	x	-
neg	b,w,l	x	x	x	x	x	x	-
negx	b,w,l	x	x	x	x	x	x	-
nop	unsized	x	x	x	x	x	x	-
not	b,w,l	x	x	x	x	x	x	-
or	b,w,l	x	x	x	x	x	x	-
ori	b,w,l	x	x	x	x	x	x	-
pack	unsized	-	x	x	x	x	x	-
pea	l	x	x	x	x	x	x	-
pflush~	unsized	-	-	x	x	x	x	-
pflusha~	unsized	-	-	x	-	-	-	-
plpa~	unsized	-	-	-	-	x	-	-
pload~	unsized	-	-	x	-	-	-	-
pmove~	w,l,q	-	-	x	-	-	-	-
ptest~	unsized	-	-	x	x	-	-	-
reset~	unsized	x	x	x	x	x	x	-
rol	b,w,l	x	x	x	x	x	x	-
ror	b,w,l	x	x	x	x	x	x	-
roxl	b,w,l	x	x	x	x	x	x	-
roxr	b,w,l	x	x	x	x	x	x	-
rtd	unsized	x	x	x	x	x	x	-
rte~	unsized	x	x	x	x	x	x	-
rtr	unsized	x	x	x	x	x	x	-
rts	unsized	x	x	x	x	x	x	-
rtm	unsized	-	x	-	-	-	-	-
sbed	b	x	x	x	x	x	x	-
scc	b	x	x	x	x	x	x	-
stop~	unsized	x	x	x	x	x	x	-
sub	b,w,l	x	x	x	x	x	x	-
subq	b,w,l	x	x	x	x	x	x	-
suba	w,l	x	x	x	x	x	x	-
subi	b,w,l	x	x	x	x	x	x	-
subx	b,w,l	x	x	x	x	x	x	-
swap	w	x	x	x	x	x	x	-
tas	b	x	x	x	x	x	x	-
trap	unsized	x	x	x	x	x	x	-
		68000	68010	68020	68030	68040	68060	6888x

		ПРИЛОЖЕНИЯ					
Операция	Размер	68000	68010	68020	68030	68040	68060
trapcc	? ,w,l	-	x	x	x	x	-
trapv	unsized x	x	x	x	x	x	-
tst	b,w,l x	x	x	x	x	x	-
unlk	unsized x	x	x	x	x	x	-
unpk	unsized -	-	x	x	x	x	-
		68000	68010	68020	68030	68040	68060

4. Сводка команд, появившихся в процессоре 68020 и операции с сопроцессором

МНЕМОНИКА	ОПИСАНИЕ
BFCHG	Проверка и инвертирование битов поля
BFCLR	Проверка и сброс битов поля
BFEXTS	Загрузка битового поля с распространением знака
BFEXTU	Загрузка битового поля без распространения знака
BFFFO	Поиск первой единицы в битовом поле
BFINS	Запись в битовое поле
BFSET	Установка битов поля
BFTST	Проверка битового поля
BKPT	Точка останова
CALLM	Вызов модуля (only 68020)
CHK2	Контроль попадания в диапазон
CMP2	Сравнение
DIVSL	Деление со знаком
DIVUL	Деление без знака
EXTB	Распространение знака
ILLEGAL	Возбуждение исключения по недопустимой команде
MOVEC	Пересылка управляющего регистра
MOVES	Пересылка между адресными пространствами
PACK	Упаковка
RTD	Возврат и освобождение области параметров
RTM	Возврат из модуля (only 68020)
TRAPcc	Условная программная ловушка
UNPK	Распаковка

ПРИЛОЖЕНИЯ

Сопроцессорные команды

срBcc	Переход по сопроцессорному условию
срDBcc	Проверка сопроцессорного условия с учетом кратности и переход
срGEN	Сопроцессорные функции общего назначения
срRESTORE	Восстановление состояния сопроцессора
срSAVE	Сохранение состояния сопроцессора
срScc	Установка по сопроцессорному условию
срTRAPcc	Ловушка по сопроцессорному условию

5. Несколько слов о процессоре 68060

За счет оптимизации скорости процессор 68060 лишен некоторых присущих предыдущим процессорам операций, что хорошо видно из таблицы. Список не реализованных целочисленных операций:

DIVU.L	<ea>,Dr:Dq	64/32 => 32r,32q
DIVS.L	<ea>,Dr:Dq	64/32 => 32r,32q
MULU.L	<ea>,Dr:Dq	32*32 => 64
MULS.L	<ea>,Dr:Dq	32*32 => 64
MOVEP	Dx,(d16,Ay)	size = W или L
MOVEP	(d16,Ay),Dx	size = W или L
CHK2	<ea>,Rn	size = B,W или L
CMP2	<ea>,Rn	size = B,W или L
CAS2	Dc1:Dc2,Du1:Du2,(Rn1):(Rn2)	size = W или L
CAS	Dc,Du,<ea>	size = W или L, невывороченный <ea>

Попытка исполнения каждой такой целочисленной инструкции вызывает Exception (исключение) по вектору 61.

Список FPU-операций, эффективного адреса и типов данных, не реализованных в 68060:

FACOS	FCOSH	FLOG10
FASIN	FETOX	FLOGN
FATAN	FETOXM1	FLOGNP1
FATANH	FGETEXP	FMOVECR
FCOS	FGETMAN	FSIN

ПРИЛОЖЕНИЯ

FSINCOS	FTWOTOX	FTRAPcc
FSINH	FLOG2	FDBcc
FTAN	FMOD	FScC
FTANH	FREM	
FTENTOX	FSCALE	

Попытка исполнения каждой такой инструкции вызывает Exception (исключение) по вектору 11.

FMOVE.M.X (dynamic register list)

FMOVE.M.L #immediate of 2 or 3 control regs

F<op>.X #immediate.FPn

F<op>.P #immediate.FPn

Попытка исполнения каждой такой инструкции вызывает Exception (исключение) по вектору 60.

Data formats	SGL	DBL	EXT	DEC	Byte	Word	Long
/Data types							
Normalized	S	S	S	U	S	S	S
Zero	S	S	S	U	S	S	S
Infinity	S	S	S	U	-	-	-
NAN	S	S	S	U	-	-	-
Denormalized	U	U	U	U	-	-	-
Unnormalized	-	-	U	U	-	-	-

Обозначения:

S - формат данных, поддерживаемый 68060

U - нереализованный формат данных, поддерживаемый пакетом софтверной эмуляции

Попытка исполнения каждой инструкции с нереализованным форматом данных вызывает Exception (исключение) по вектору 55.

Все перечисленные выше инструкции и форматы данных эмулируются программно, прилагаемым к 68060 акселераторам программным обеспечением. Операция эмуляции проходит следующим образом:

* процессор обнаруживает неподдерживаемую

ПРИЛОЖЕНИЯ

- команду
- * ожидает окончания исполнения предыдущих инструкций (находящихся на конвейере)
- * происходит собственно соответствующее Exception (исключение).

Таким образом, процессор начинает работать "скачками", отчего зффективность суперскалярности (параллельности исполнения команд, предсказания и т.д.) резко снижается ...

Инструкции, идущие ниже, **ВООБЩЕ** не поддерживаются пакетом софтверной эмуляции команд M68K фирмы Motorola, в том числе и 68060.library, поэтому их использовать нельзя.

Инструкция	Адресация
DIV{U,S}.L	-(SSP),DR:DQ
MUL{U,S}.L	-(SSP),DR:DQ
F<OP>.P	-(SSP),FPN
F<OP>.P	FPN,(SSP)+
F<OP>.{B,W,L,S,D,X}	-(SSP),FPN
FS<CC>.B	-(SSP)
FMOVEM.X	-(SSP),DN
FMOVEM.X	DN,(SSP)+
F<OP>.X	FPN,(SSP)+
F<OP>.{B,W,L}	FPN,(SSP)+

Очевидно, что использование этих инструкций вызовет полную неработоспособность программы на 68060.

6. Расчет времени выполнения инструкций процессора 68000

Для расчета времени выполнения большинства инструкций 68000, необходимо вначале найти количество циклов, используемое методом адресации в таблице, приведенной ниже, а затем время выполнения инструкции в соответствующей таблице.

В данной таблице приведены значения количества периодов, требуемых для вычисления эффективного адреса

ПРИЛОЖЕНИЯ

инструкции. Это количество включает время выборки любых расширенных слов, вычисление адреса, выборку операнда из памяти. Количество циклов чтения и записи шины показано соответственно как (чтение/запись).
Примечание: циклы записи не включены в вычисление эффективного адреса.

Время вычисления эффективного адреса

	Регистровая адресация	byte, word	long
Dn	прямая по регистру данных	0(0/0)	0(0/0)
An	прямая по регистру адреса	0(0/0)	0(0/0)
	Память		
(An)	регистровая косвенная простая	4(1/0)	8(2/0)
(An)+	регистровая косвенная с пост-инкрементом	4(1/0)	8(2/0)
-(An)	регистровая косвенная с предкрементом	6(1/0)	10(2/0)
d(An)	регистровая косвенная со смещением	8(2/0)	12(3/0)
d(An,ix)	регистровая косвенная с индексацией	10(2/0)	14(3/0)
xxx.W	абсолютная короткая	8(2/0)	12(3/0)
xxx.L	абсолютная длинная	12(3/0)	16(4/0)
d(PC)	счетчик команд со смещением	8(2/0)	12(3/0)
d(PC,ix)	счетчик команд с индексацией	10(2/0)	14(3/0)
#xxx	непосредственная	4(1/0)	8(2/0)

Размер индексного регистра не влияет на время выполнения.

ВРЕМЯ ВЫПОЛНЕНИЯ ИНСТРУКЦИЙ ПЕРЕСЫЛКИ

Следующие две таблицы показывают количество периодов для инструкции MOVE. Эти данные включают выборку инструкции, чтение операндов и запись операндов. Количество циклов чтения и записи шины показаны соответственно как (чтение/запись).

Время выполнения инструкций MOVE.B и MOVE.W приведено в таблице.

ПРИЛОЖЕНИЯ

	Dn	An	(An)	(An) +	-(An)	d(An)	d(An,ix)	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	0(1/1)	0(1/1)	0(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
An	4(1/0)	4(1/0)	8(1/1)	0(1/1)	8(1/1)	12(2/1)	14(2/1)	12(2/1)	16(3/1)
(An)	8(2/0)	0(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	10(3/1)	16(3/1)	20(4/1)
(An) +	0(2/0)	8(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	18(3/1)	16(3/1)	20(4/1)
-(An)	10(2/0)	10(2/0)	14(2/1)	14(2/1)	14(2/1)	18(3/1)	20(4/1)	18(3/1)	22(4/1)
d(An)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(An,ix)	14(3/0)	14(3/0)	10(3/1)	10(3/1)	10(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
xxx.W	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
xxx.L	16(4/0)	16(4/0)	20(4/1)	20(4/1)	20(4/1)	24(5/1)	26(5/1)	24(5/1)	20(6/1)
d(PC)	12(3/0)	12(3/0)	16(3/1)	16(3/1)	16(3/1)	20(4/1)	22(4/1)	20(4/1)	24(5/1)
d(PC,ix)	14(3/0)	14(3/0)	10(3/1)	10(3/1)	10(3/1)	22(4/1)	24(4/1)	22(4/1)	26(5/1)
#xxx	0(2/0)	0(2/0)	12(2/1)	12(2/1)	12(2/1)	16(3/1)	10(3/1)	16(3/1)	20(4/1)

Размер индексного регистра не влияет на время выполнения.

Время выполнения инструкций MOVE.L

	Dn	An	(An)	(An) +	-(An)	d(An)	d(An,ix)	xxx.W	xxx.L
Dn	4(1/0)	4(1/0)	12(1/2)	12(1/2)	12(1/2)	16(2/2)	10(2/2)	16(2/2)	20(3/2)
An	4(1/0)	4(1/0)	12(1/2)	12(1/2)	12(1/2)	16(2/2)	10(2/2)	16(2/2)	20(3/2)
(An)	12(3/0)	12(3/0)	20(3/2)	20(3/2)	20(3/2)	24(4/2)	26(4/2)	24(4/2)	28(5/2)
(An) +	12(3/0)	12(3/0)	20(3/2)	20(3/2)	20(3/2)	24(4/2)	26(4/2)	24(4/2)	28(5/2)
-(An)	14(3/0)	14(3/0)	22(3/2)	22(3/2)	22(3/2)	26(4/2)	28(4/2)	26(4/2)	30(5/2)
d(An)	16(4/0)	16(4/0)	24(4/2)	24(4/2)	24(4/2)	20(5/2)	30(5/2)	28(5/2)	32(6/2)
d(An,ix)	18(4/0)	10(4/0)	26(4/2)	26(4/2)	26(4/2)	30(5/2)	32(5/2)	30(5/2)	34(6/2)
xxx.W	16(4/0)	16(4/0)	24(4/2)	24(4/2)	24(4/2)	20(5/2)	30(5/2)	20(5/2)	32(6/2)
xxx.L	20(5/0)	20(5/0)	20(5/2)	20(5/2)	20(5/2)	32(6/2)	34(6/2)	32(6/2)	36(7/2)
d(PC)	16(4/0)	16(4/0)	24(4/2)	24(4/2)	24(4/2)	20(5/2)	30(5/2)	20(5/2)	32(5/2)
d(PC,ix)	10(4/0)	18(4/0)	26(4/2)	26(4/2)	26(4/2)	30(5/2)	32(5/2)	30(5/2)	34(6/2)
#xxx	12(3/0)	12(3/0)	20(3/2)	20(3/2)	20(3/2)	24(4/2)	26(4/2)	24(4/2)	28(5/2)

Размер индексного регистра не влияет на время выполнения.

ВРЕМЯ ВЫПОЛНЕНИЯ ОБЫЧНЫХ ИНСТРУКЦИЙ

Количество периодов, показанных в этой таблице, отражает время, требуемое для выполнения операций, сохранения результатов, и чтения следующей инструкции. Количество циклов чтения и записи шины показано

ПРИЛОЖЕНИЯ

соответственно как (чтение/запись). Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Заголовки таблицы означают:

Аn и Dn - регистровые операнды,

ea - операнд определяемый эффективным адресом,

M - операнд в памяти.

Инструкция	Размер	op<ea>,An ^	op<ea>,Dn	op Dn,<M>
ADD	byte,word	8(1/0) +	4(1/0) +	8(1/1) +
	long	6(1/0) +**	6(1/0) +**	12(1/2) +
AND	byte,word	-	4(1/0) +	8(1/1) +
	long	-	6(1/0) +**	12(1/2) +
CMP	byte,word	6(1/0) +	4(1/0) +	-
	long	6(1/0) +	6(1/0) +	-
DIVS	-	-	158(1/0) +*	-
DIVU	-	-	140(1/0) +*	-
EOR	byte,word	-	4(1/0) ***	8(1/1) +
	long	-	8(1/0) ***	12(1/2) +
MULS	-	-	70(1/0) +*	-
MULU	-	-	70(1/0) +*	-
OR	byte,word	-	4(1/0) +**	8(1/1) +
	long	-	6(1/0) +**	12(1/2) +
SUB	byte,word	8(1/0) +	4(1/0) +	8(1/1) +
	long	6(1/0) +**	6(1/0) +**	12(1/2) +

Обозначения:

+ добавляется время вычисления эффективного адреса

^ Только word или long

* показывает максимальное значение

** базовое время в шесть периодов увеличивается до восьми, если адресация - прямая регистровая или непосредственная (время эффективного адреса тоже добавляется)

ПРИЛОЖЕНИЯ

*** единственная доступная адресация - прямая регистровая

DIVS,

DIVU время выполнения алгоритма деления, использованного в 68000, для лучшего и худшего вариантов различается менее чем на 10 процентов.

MULS,

MULU алгоритм умножения требует $38+2n$ периодов, где n :

MULU: n = число периодов в <ea>

MULS: n = сочетание <ea> с нулем в качестве младшего значащего бита; n - количество сочетаний 10 или 01 в 17-битном источнике, то есть, худший вариант - если источник = \$5555

ВРЕМЯ ВЫПОЛНЕНИЯ НЕПОСРЕДСТВЕННЫХ ИНСТРУКЦИЙ

Количество периодов, показанное в этой таблице, включает время выборки непосредственных операндов, выполнения операций, сохранения результатов и чтения следующей операции. Количество циклов чтения и записи шины показано соответственно как (чтение/запись). Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Инструкция	Размер	op #, Dn	op #, An	op #, M
ADDI	byte, word	8(2/0)	-	12(2/1) +
	long	16(3/0)	-	20(3/2) +
ADDQ	byte, word	4(1/0)	8(1/0) *	8(1/1) +
	long	8(1/0)	8(1/0)	12(1/2) +
ANDI	byte, word	8(2/0)	-	12(2/1) +
	long	16(3/0)	-	20(3/1) +
CMPI	byte, word	8(2/0)	-	8(2/0) +
	long	14(3/0)	-	12(3/0) +
EORI	byte, word	8(2/0)	-	12(2/1) +

ПРИЛОЖЕНИЯ

	long	16(3/0)	-	20(3/2) +
MOVEQ	long	4(1/0)	-	-
ORI	byte,word	8(2/0)	-	12(2/1) +
	long	16(3/0)	-	20(3/2) +
SUBI	byte,word	8(2/0)	-	12(2/1) +
	long	16(3/0)	-	20(3/2) +
SUBQ	byte,word	4(1/0)	8(1/0) *	8(1/1) +
	long	8(1/0)	8(1/0)	12(1/2) +

Обозначения:

- + добавляется время вычисления эффективного адреса
- * только word

ВРЕМЯ ВЫПОЛНЕНИЯ ОДНООПЕРАНДНЫХ ИНСТРУКЦИЙ

В этой таблице приведено количество периодов для инструкций с одним операндом. Количество циклов чтения и записи шины показано соответственно как (чтение/запись). Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Инструкция	Размер	Регистр	Память
CLR	byte,word	4(1/0)	8(1/1) +
	long	6(1/0)	12(1/2) +
NBCD	byte	6(1/0)	8(1/1) +
NEG	byte,word	4(1/0)	8(1/1) +
	long	6(1/0)	12(1/2) +
NEGX	byte,word	4(1/0)	8(1/1) +
	long	6(1/0)	12(1/2) +
NOT	byte,word	4(1/0)	8(1/1) +
	long	6(1/0)	12(1/2) +
Sec	byte,false	4(1/0)	8(1/1) +
	byte,true	8(1/0)	8(1/1) +

ПРИЛОЖЕНИЯ

TAS #	byte	4(1/0)	10(1/1) +
TST	byte, word	4(1/0)	4(1/0) +
	long	4(1/0)	4(1/0) +

Обозначения:

- +** добавляется время вычисления эффективного адреса.
- #** Эту инструкцию нельзя использовать - ее цикл чтения/записи может нарушить работу DMA.

ВРЕМЯ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ СДВИГА

В этой таблице приведено количество периодов для инструкций сдвига. Количество циклов чтения и записи шины показано соответственно как (чтение/запись). Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Инструкция	Размер	Регистр	Память
ASR, ASL	byte, word	6+2n(1/0)	8(1/1) +
	long	8+2n(1/0)	-
LSR, LSL	byte, word	6+2n(1/0)	8(1/1) +
	long	8+2n(1/0)	-
ROR, ROL	byte, word	6+2n(1/0)	8(1/1) +
	long	8+2n(1/0)	-
ROXR, ROXI	byte, word	6+2n(1/0)	8(1/1) +
	long	8+2n(1/0)	-

Обозначения:

- +** добавляется время вычисления эффективного адреса
- n** счетчик сдвига

ВРЕМЯ ВЫПОЛНЕНИЯ БИТОВЫХ ОПЕРАЦИЙ

В этой таблице приведено количество периодов для инструкций управления битами. Количество циклов чтения и записи шины показано соответственно как

ПРИЛОЖЕНИЯ

(чтение/запись). Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Инструкция	Размер	dynamic		static	
		регистр	память	регистр	память
BCHG	byte	-	8(1/1) +	-	12(2/1) +
	long	8(1/0) *	-	12(2/0) *	-
BCLR	byte	-	8(1/1) +	-	12(2/1) +
	long	10(1/0) *	-	14(2/0) *	-
BSET	byte	-	8(1/1) +	-	12(2/1) +
	long	8(1/0) *	-	12(2/0) *	-
BTST	byte	-	4(1/0) +	-	8(2/0) +
	long	6(1/0)	-	10(2/0)	-

Обозначения:

- + добавляется время вычисления эффективного адреса
- * показывает максимальное значение

ВРЕМЯ ВЫПОЛНЕНИЯ УСЛОВНЫХ ИНСТРУКЦИЙ

В этой таблице приведено количество периодов для условных инструкций. Количество циклов чтения и записи шины показано соответственно как (чтение/запись).

Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Инструкция	Смещение	Переход был	Перехода не было
Bcc	byte	10(2/0)	8(1/0)
	word	10(2/0)	12(1/0)
BRA	byte	10(2/0)	-
	word	10(2/0)	-
BSR	byte	18(2/2)	-
	word	18(2/2)	-

ПРИЛОЖЕНИЯ

DBcc	CC true -	12(2/0)
	CC false 10(2/0)	14(3/0)

ВРЕМЯ ВЫПОЛНЕНИЯ ИНСТРУКЦИЙ JMP, JSR, LEA, PEA И MOVEM

В этой таблице приведено количество периодов для инструкций JMP, JSR, LEA, PEA и MOVEM. Количество циклов чтения и записи шины показано соответственно как (чтение/запись). Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Инстр.	Размер	(An)	(An)+	-(An)	d(An)
JMP	-	8(2/0)	-	-	10(2/0)
JSR	-	16(2/2)	-	-	18(2/2)
EA	-	4(1/0)	-	-	8(2/0)
PEA	-	12(1/2)	-	-	16(2/2)
MOVEM	word	12+4n	12+4n	-	16+4n
M->R		(3+n/0)	(3+n/0)	-	(4+n/0)
	long	12+8n	12+8n	-	16+8n
		(3+2n/0)	(3+2n/0)	-	(4+2n/0)
MOVEM	word	8+4n	-	8+4n	12+4n
R->M		(2/n)	-	(2/n)	(3/n)
	long	8+8n	-	8+8n	12+8n
		(2/2n)	-	(2/2n)	(3/2n)

Инстр.	Размер	d(A ₁ ,ix)+	xxx.W	xxx.L	d(PC)	d(PC,ix)*
JMP	-	14(3/0)	10(2/0)	12(3/0)	10(2/0)	14(3/0)
JSR	-	22(2/2)	18(2/2)	20(3/2)	18(2/2)	22(2/2)
LEA	-	12(2/0)	8(2/0)	12(3/0)	8(2/0)	12(2/0)
PEA	-	20(2/2)	16(2/2)	20(3/2)	16(2/2)	20(2/2)
MOVEM	word	18+4n	16+4n	20+4n	16+4n	18+4n
M->R		(4+n/0)	(4+n/0)	(5+n/0)	(4+n/0)	(4+n/0)
	long	18+8n	16+8n	20+8n	16+8n	18+8n

ПРИЛОЖЕНИЯ

		$(4+2n/0)$	$(4+2n/0)$	$(5+2n/0)$	$(4+2n/0)$	$(4+2n/0)$
MOVEM	word	$14+4n$	$12+4n$	$16+4n$	-	-
R->M		$(3/n)$	$(3/n)$	$(4/n)$	-	-
	long	$14+8n$	$12+8n$	$16+8n$	-	-
		$(3/2n)$	$(3/2n)$	$(4/2n)$	-	-

Обозначения:

n количество регистров

* размер индексного регистра не влияет на время выполнения

ВРЕМЯ ВЫПОЛНЕНИЯ ОПЕРАЦИЙ С ПОВЫШЕННОЙ ТОЧНОСТЬЮ

В этой таблице приведено количество периодов для операций повышенной точности. Количество циклов чтения и записи шины показано соответственно как (чтение/запись). Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Заголовок таблицы: Dn - регистр данных, M - память.

Инструкция	Размер	op Dn,Dn	op M,M
ADDX	byte,word	4(1/0)	18(3/1)
	long	8(1/0)	30(5/2)
CMPM	byte,word	-	12(3/0)
	long	-	20(5/0)
SUBX	byte,word	4(1/0)	18(3/1)
	long	8(1/0)	30(5/2)
ABCD	byte	6(1/0)	18(3/1)
SBCD	byte	6(1/0)	18(3/1)

ВРЕМЯ ВЫПОЛНЕНИЯ ДРУГИХ РАЗЛИЧНЫХ ИНСТРУКЦИЙ

В этой таблице приведено количество периодов для различных инструкций. Количество циклов чтения и записи шины показано соответственно как (чтение/запись).

ПРИЛОЖЕНИЯ

Количество периодов и циклов чтения/записи должно быть добавлено соответственно к показанным значениям времени вычисления эффективного адреса.

Инструкция	Размер	Регистр	Память
ANDI to CCR	byte	20(3/0)	-
ANDI to SR	word	20(3/0)	-
CHK	-	10(1/0) +	-
EORI to CCR	byte	20(3/0)	-
EORI to SR	word	20(3/0)	-
ORI to CCR	byte	20(3/0)	-
ORI to SR	word	20(3/0)	-
MOVE from SR	-	6(1/0)	3(1/1)+
MOVE to CCR	-	12(1/0)	12(1/0)+
MOVE to SR	-	12(1/0)	12(1/0)+
EXG	-	6(1/0)	-
EXT	word	4(1/0)	-
	long	4(1/0)	-
LINK	-	16(2/2)	-
MOVE from USP	-	4(1/0)	-
MOVE to USP	-	4(1/0)	-
NOP	-	4(1/0)	-
RESET	-	132(1/0)	-
RTE	-	20(5/0)	-
RTR	-	20(5/0)	-
RTS	-	16(4/0)	-
STOP	-	4(0/0)	-
SWAP	-	4(1/0)	-
TRAPV (No Trap)	-	4(1/0)	-
UNLK	-	12(3/0)	-

Обозначения:

+ добавляется время вычисления эффективного адреса

ПРИЛОЖЕНИЯ

ВРЕМЯ ВЫПОЛНЕНИЯ ИНСТРУКЦИЙ ОБМЕНА С ПЕРИФЕРИЕЙ

<u>Инструкция</u>	<u>Размер</u>	<u>Регистр-память</u>	<u>Память-регистр</u>
MOVEP	word	16(2/2)	16(4/0)
	long	24(2/4)	24(6/0)

ВРЕМЯ ОБРАБОТКИ ПРЕРЫВАНИЙ

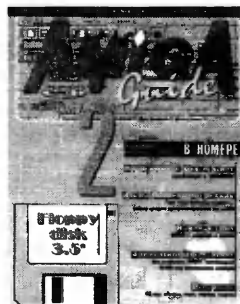
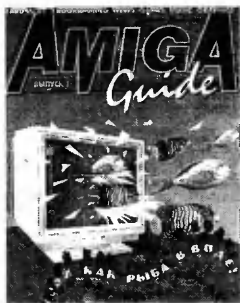
В данной таблице приведено количество периодов для обработки прерываний. Число периодов включает время работы со стеком, выборки вектора и выборки первых двух слов инструкций процедуры обработки прерывания. Количество циклов чтения и записи шины показано соответственно как (чтение/запись).

<u>Прерывание</u>	<u>Периоды</u>
ошибка адреса	50(4/7)
ошибка шины	50(4/7)
операция CHK (было прерывание)	44(5/3)+
деление на ноль	42(5/3)
неверная инструкция	34(4/3)
прерывание	44(5/3)*
нарушение привилегий	34(4/3)
RESET **	40(6/0)
трассировка	34(4/3)
инструкция TRAP	38(4/3)
инструкция TRAPV (было прерывание)	34(4/3)

Обозначения:

- + добавляется время вычисления эффективного адреса
- * цикл подтверждения прерывания занимает четыре периода
- ** время от RESET и HALT до начала выполнения инструкций

ООО "ФОРМАК" имеет честь представить Вам первый русскоязычный журнал полностью посвященный проблематике компьютера AMIGA. На его страницах мы постараемся обобщить опыт как зарубежных, так и отечественных пользователей и программистов.



Основные разделы журнала:

- **раздел для начинающих**, в котором представлены начальные сведения по истории развития компьютера, его функциональным возможностям, основам программирования и т.п.;
- **раздел "Железные правила"**, посвященный архитектуре компьютера, распределению памяти, системе команд процессора, протоколам обмена информацией аппаратным новинкам и возможным доработкам компьютера...
- **раздел "Программное обеспечение"** — это описание как самых новых, так и самых популярных системных и прикладных программ;
- **раздел "Программистское чтение"**, в котором представлены описания различных языков программирования, наиболее интересные процедуры, нестандартные решения;
- **раздел "ИГРОДРОМ"** с

описаниями новейших и любимейших программ, секреты к играм, хитрости, пароли, коды и другая бесценная информация для геймеров.

САМОЕ ГЛАВНОЕ: каждый из вас может стать полноправным соавтором журнала. Нашей основной задачей является объединение амиговцев всей страны через наш с Вами журнал. Поэтому, если у Вас есть чем поделиться с другими читателями, будь-то мощная системная программа или небольшая изящная процедура, обширное описание игры или маленький секрет к ней, схема нового периферийного оборудования или мелкое аппаратное усовершенствование — присылайте все, все, все к нам в журнал, делитесь своими достижениями, задавайте вопросы. Начинаящим мы постараемся помочь разрешить текущие проблемы, а профессионалам дадим возможность заявить о себе и стать известными. Давайте создавать наш с вами журнал вместе. На основе журнала мы планируем, также, создать мощный рынок зарубежного и отечественного программного обеспечения. Все читатели журнала смогут приобрести лично или по почте все новое и лучшее, что создано или будет создано зарубежными и отечественными программистами для Амиги. Авторам качественного программного обеспечения мы предлагаем услуги по распространению их программ с выплатой авторского вознаграждения. С удовольствием поможем объединиться многочисленным разрозненным клубам любителей Амиги. Их адреса опубликуем бесплатно. Примем в журнал и платную коммерческую рекламу.

Все, кто заинтересован в публикации своих материалов, покупке журнала и/или его распространении, оформлении подписки на 5 номеров 1997 года, в решении любых других вопросов — пишите нам по адресу: 121010, Москва, а/я 16. ТОО "ФОРМАТ" с почтой "Amiga". Для получения информации вложите конверт с Вашим адресом.

В книге рассмотрен язык ассемблера, дающий программисту возможность в полной мере использовать ресурсы и скорость компьютера и принципы программирования на нем для компьютера Amiga.

Рассмотрена организация памяти и основные функции компьютера, внутренняя структура Amiga и ее процессор, обзор команд процессора и многое другое. Описание сопровождается множеством примеров, которые, несомненно, помогут Вам лучше понять материал.

Мы надеемся, что эта книга поможет многим пользователям Amiga в изучении машинного программирования и в создании новых полезных программ.

© ТОО "ФОРМАК", 1997

AMIGA: Программирование на ассемблере

Издатель: ТОО "ФОРМАК"
ЛР 063856
от 30 декабря 1994 г.
Для писем 121019, Москва, в/я 16

Подписано в печать 04.06.97 г. Формат 84×108/32.
Печать офсетная. Объем 9,5 п. л. Тираж 1200. Зак. 336.

Отпечатано с готового оригинал-макета в типографии ИПО Профиздат,
109044, Москва, Крутицкий вал, д. 18.

AMIGA:

Программирование на ассемблере

Ассемблер - это "родной" язык Amiga, дающий программисту возможность в полной мере использовать ресурсы и скорость компьютера.

Программирование на ассемблере требует знаний процессора MC68000 и внутренних характеристик Amiga. Большое число функций, предлагаемых операционной системой (ОС), доступны и широко используются при программировании на ассемблере.

Стандартная документация по функциям ОС написана для использования в С-программах, и практически бесполезна при программировании на ассемблере. В этой книге мы рассмотрим процессор MC68000, операционную систему и ее функции с точки зрения программиста на ассемблере.

*обработал и преобразовал в DjVu
Александр Богомаз,
albom85@yandex.ru*