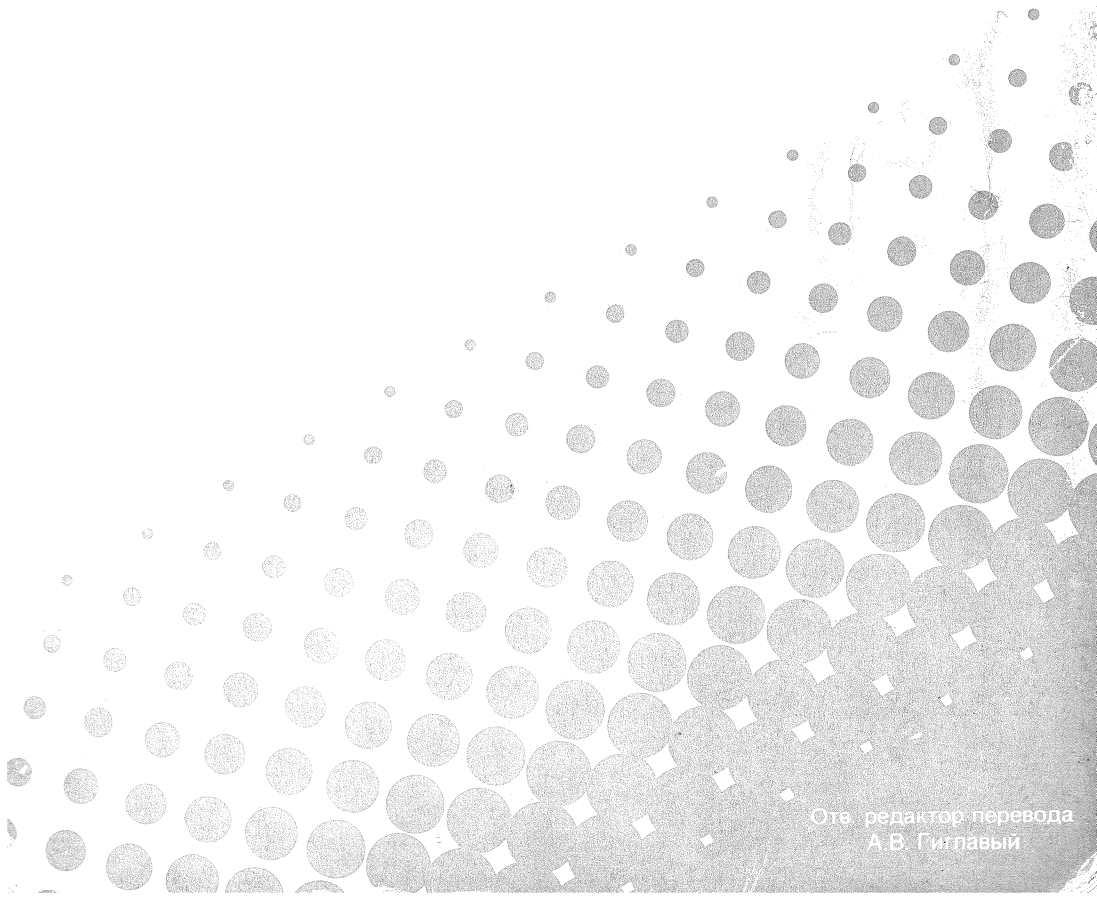
NIPPON GAKKI CO., LTD.

MSX

YAMAHA

СПРАВОЧНОЕ РУКОВОДСТВО

по языку программирования БЕЙСИК
для комплектов учебной вычислительной техники
на базе персональных компьютеров ЯМАХА
(перевод с английского)



Отв. редактор перевода
А.В. Гиглавый

YAMAHA

СПРАВОЧНОЕ РУКОВОДСТВО

по языку программирования БЕЙСИК
для комплектов учебной вычислительной техники
на базе персональных компьютеров ЯМАХА
(перевод с английского)

ПРЕДИСЛОВИЕ

Это руководство предназначено для тех пользователей КУВТ ЯМАХА, которые разрабатывают педагогические программные средства с применением языка Бейсик. Уровень изложения рассчитан на программистов-профессионалов; поэтому не рекомендуется применять материалы руководства непосредственно в качестве учебного пособия для средних школ.
При создании и использовании таких учебных пособий рекомендуется использовать существующие методические разработки для различных реализаций языка Бейсик с обязательным учетом деталей реализации MSX-Бейсик.

Текст руководства подготовлен японскими фирмами "Синдзидайся" и "ЯМАХА" в сотрудничестве с Институтом проблем информатики АН СССР.

Ввиду весьма сжатых сроков подготовки и редактирования перевода в текст вкрались неустраненные опечатки. Фирмы "ЯМАХА" и "Синдзидайся" и редактор перевода приносит за это свои извинения и намерены улучшить качество текста в последующих изданиях.

Отв. редактор перевода
А.В. Гиглавый

В данном руководстве описан язык программирования MSX-Бейсик, используемый для персональных ЭВМ Yamaha MSX Пожалуйста, внимательно прочитайте это руководство и руководство пользователя ЭВМ Yamaha MSX (Ямана КУВТ).

Программы, написанные на языке MSX-Бейсик, могут переноситься с одной ЭВМ серии MSX на другую ЭВМ этой же серии (независимо от конкретной фирмы-изготовителя) благодаря тому, что MSX-Бейсик стандартизован.

Глава 1 содержит введение в систему MSX и перечень команд, операторов и функций.

Глава 2 содержит детальное описание основных понятий языка MSX-Бейсик: режимов работы, констант, переменных, операторов, форматов данных и т.д.

Глава 3 представляет собой полное справочное руководство. Здесь описаны в алфавитном порядке все операторы, команды, функции и псевдопеременные языка Бейсик.

Глава 4 содержит рекомендации по программированию на языке MSX-Бейсик. Здесь вопросы использования свойств языка рассматриваются не в алфавитном порядке, а по назначению.

В приложениях приводятся табличный справочный метериал по дополнительным возможностям языка, а также перечень сообщений об ошибках с пояснениями.

Содержание

Глава 1. Введение в MSX

Глава 2. Общие сведения о языке MSX-Бейсик

Глава 3. Алфавитный указатель ключевых слов языка MSX-Бейсик

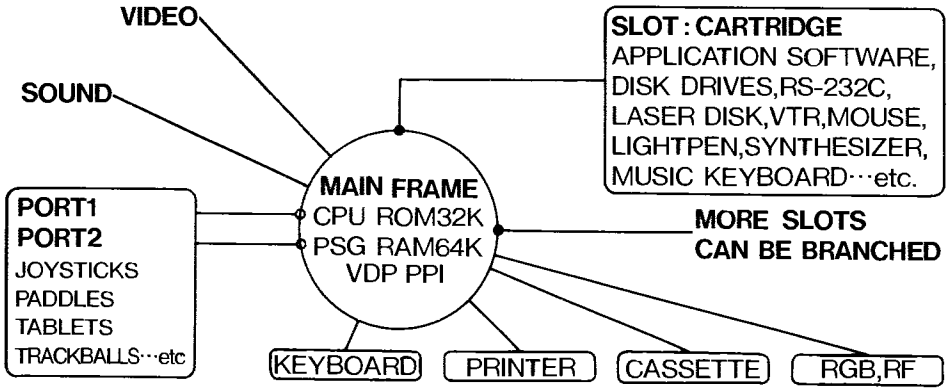
ГЛАВА 1

Введение в MSX

1.1 Аппаратные средства MSX

Аппаратные средства MSX включают: микропроцессор Z-80, работающий с тактовой частотой 3,6 Мгц; программируемый периферийный Интерфейс (PPI) Интел 8255: видео дисплейный процессор (VDP), обеспечивающий работу со спрайтами (см. раздел 4.2.3) и с видео-памятью — VRAM; программируемый генератор звуков Дженерал Инструментс AY-2-8910. Кроме того, для работы MSX требуется 32 Кбайт ПЗУ, как минимум 16 Кбайт ОЗУ и 16 Кбайт видео-памяти (VRAM). предназначенной исключительно для работы VDP. Эти Интерфейсные микросхемы (или аналогичные им по функциям) должны иметь заранее заданные и жестко зафиксированные номера портов ввода/вывода.

Для подключения извне дополнительных устройсть имеется шина расширителей магистрали, подключенная как минимум к одному выведенному наружу разъему. Процессор может единовременно адресовать 64 К байт памяти, разбитой на страницы по 16 К байт в каждой, но с помощью расширителей магистрали процессор может работать с памятью объемом до 1 М байт.



К расширителям магисграли могут подключаться кассеты ПЗУ или другие расширители магистрали для расширения возможностей ЭВМ.

1.2 Встроенное программное обеспечение MSX

Встроенное программное обеспечение записано в ПЗУ. В первую очередь к нему относится интерпретатор языка MSX-Бейсик. Этот язык представляет собой значительно расширенную версию языка Бейсик фирмы "Микрософт." Он допускает работу в графическом режиме, обработку прерываний, выдачу звуковых сигналов, а также доступ ко всем другим возможностям, имеющимся в микро ЭВМ.

Еще одним свойством языка MSX-Бейсик является его расширяемость. С помощью специального оператора CALL можно добавлять новые операторы. Можно задавать имена новых устройств ввода/вывода (в/в). И, наконец, с помощью более чем 100 стандартизованных "перехватов", можно из ОЗУ прерывать выполнение программ, хранящихся в ПЗУ. Это предоставляет практически неограниченные возможности для расширения аппаратных и программных средств. Интерпретатор MSX-Бейсик опознает подключенные кассеты ПЗУ и позволяет выполнить хранящиеся в них программы на языке MSX-Бейсик или в машинных кодах.

Для программистов, программирующих в машинных кодах (или для программистов, применяющих различные ухищрения в Бейсик-программах), имеется более 100 точек входа в MSX-Бейсик.

Этот набор точек определяется содержимым системного ПЗУ (базовая система в/в, BIOS). Хранящиеся в ОЗУ системные переменные, используемые интерпретатором языка MSX-Бейсик и программами BIOS также стандартизованы и доступны для программистов.

Доступ к этим переменным этим переменным делает возможным использование многих программистских приемов, невозможных в обычном языке Бейсик, причем гарантируется совместимость с новыми версиями MSX, которые появятся в ближайшем будущем.

1.3 Классификация команд, операторов и функции

Одним из удобных средств в интерпретаторе MSX-Бейсик является библиотека подпрограмм для математических вычислений. Этот пакет производит вычисления с целыми числами, с вещественными числами одинарной точности (до 6 цифр), с числами двойной точности (до 14 цифр), а также с двоично-десятичными числами. Это означает, что при выполнении арифметических действий не возникнут значительные ошибки, вызванные округлением, как, например в языке GW-Бейсик (диалект для IBM pc — прим. ред.). Вычисление функций ведется с точностью, гарантирующей минимальные ошибки.

Система MSX обеспечивает возможность работы с кассетным магнитофоном со скоростью 2400 бод, что позволяет передавать 240 знаков в секунду — это максимально возможная в настоящее время скорость работы с магнитофонной лентой. Имеются также возможности работы со строками символов и средства экранного редактирования.

В последующих разделах подробно описаны все типы команд, операторов, функций и псевдопеременных, имеющихся в MSX-Бейсик. Команды обычно вводятся в прямом, или непосредственном, режиме, т.е. выполняются сразу после ввода.

Функции можно опознать по наличию круглых скобок ( ). Функции выполняют вычисления и возвращают результат.

Псевдопеременные — это специальные функции. Они могут либо передавать, либо получать значения. Наличие псевдопеременных облегчает доступ к некоторым ресурсам машины.

Операторы приведены ниже в алфавитном порядке (для удобства изучения все операторы обьединены в группы).

1.3.1 Изменение текста программ в памяти

AUTO	автоматическая нумерация строк для ускорения ввода текста
DELETE	удаление группы строк из памяти
LIST	вывод программы на экран

LLIST	вывод программы на печатающее устройство
NEW	удаление всей программы из памяти и выполнение оператор CLEAR
RENUM	перенумерация строк в памяти

1.3.2 Команды чтения и записи на ленту

BSAVE	запись блока памяти на магнитную ленту в двоичном формате
BLOAD	чтение информации с магнитеной ленты в ОЗУ в двоичном формате
CLOAD	загрузка программы с ленты; программа должна быть записана во внутреннем формате
CSAVE	запись программы на ленту во внутреннем формате
LOAD	загрузка программы с ленты в формате ASCII
MERGE	добавление программы с ленты в формате ASCII к программе, находящейся в ОЗУ
RUN “ ”	загрузка и запуск программы с ленты в формате ASCII
SAVE	запись программы на ленту в формате ASCII

1.3.3 Инициализация некоторых основных средств

CLEAR	резервирует ОЗУ для строковых массивов и/или использования не для нужд Бейсика; одновременно инициализирует все переменные, открытые файлы и т.д.
KEY	переопределяет использование специальных клавиш; KEY LIST выводит на экран список определений для всех 10 функциональных клавиш

SCREEN задает режим работы экрана. Устанавливает также некоторые машинно-зависимые ключи

TRON выводит на экран номера выполняемых строк (трассу)

TROFF прекращает вывод на экран номеров строк программы во время выполнения

1.3.4 Управление выполнением программ

CONT продолжает выполнение после оператора STOP или нажатия клавиш Control-STOP

RUN начинает выполнение программы

1.3.5 Начальная загрузка изменяемых параметров

CLEAR определяет размер области ОЗУ для символьных строк и области, не используемой в Бейсике; инициализирует переменные

DEF FN описывает функции, определяемые пользователем

DEF INT/SNG/DBL/STR определяет ТИП переменной, принимаемый по умолчанию

DEF USR определяет начальные адреса функций, написанных пользователем на машинном языке

DIM определяет, резервирует и инициализирует массивы

ERASE стирает указанный массив

MAXFILES указывает количество файлов в/в, что необходимо для резервирования рабочих областей в ОЗУ

1.3.6 Изменение значений переменных

FOR оператор цикла; изменяет значение переменной цикла

LET оператор присваивания (использование слова LET необязательно)

SWAP обмен значениями между двумя переменными

1.3.7 Числовые функции

ABS ( ) абсолютное значение

ATN ( ) арктангенс, функция обратная тангенсу; аргументом является угол, заданный в радианах

CINT ( ) перевод целых чисел в 16-битное представление со знаком

CDBL ( ) перевод чисел в формат с двойной точностью

CSNG ( ) перевод в формат с одинарной точностью

COS ( ) косинус угла в радианах

EXP ( ) е возводится в указанную степень

FIX ( ) Отбрасывай дробнуло гась, возращай толвко целое

FRE ( ) дает сведения об оставшемся свободным пространстве в ОЗУ

INT ( ) выдает целое, ближайшее (снизу) к данному числу

LOG ( ) натуральный логарифм

RND ( ) генератор псевдослучайных чисел

SGN ( ) функция определения знака числа

SIN ( ) синус угла в радианах

SQR ( ) квадратный корень

TAN ( ) тангенс угла в радианах

1.3.8 Строковые функции

ASC ( ) выдает число, соответствующее коду заданного символа в таблице MSX-ASCII (вариант КУВТ)

BIN$ ( ) создает символьную строку, содержащую двоичное представление данного числа

CHR$ ( ) создает символ, соответствующий данному числу (коду)

HEX$ ( ) создает строку, содержащую шестнадцатеричное представление данного числа

INSTR ( ) поиск подстроки в строке

LEFT$ ( ) выделение левой части строки

LEN ( ) определяет длину строки

MID$ ( ) выделение подстроки из любого места строки

MID$ ( )= заменяет текущее содержание подстроки в данной строке

OCT$ ( ) создает строку, содержащую шестнадцатеричное представление данного числа

RIGHT$ ( ) выделение правой части строки

SPACE$ ( ) создает строку любой длины, содержащую пробелы

SPC ( ) функция для PRINT, печатающая пробелы

STR$ ( ) преобразует число в символьную строку, содержащую это число

STRING$ ( ) создает строку с повторяющимися символами

TAB ( ) функция табуляции для PRINT, печатающая пробелы вплоть до указанной позиции

VAL ( ) переводит символьную строку, содержащую цифры, в число

1.3.9 Изменение порядка программы

CALL обращение к дополнительным операторам, включаемым в словарь Бейсика

END заканчивает выполнение программы; переводит Бейсик в прямой режим

FOR начинает многократное выполнение некоторой последовательности операторов

GOSUB вызывает подпрограмму

GOTO переходит к строке с заданным номером

IF/THEN/ ELSE выбирает различные точки перехода в зависимости от условия

NEXT заканчивает цикл выполнения , начатый оператором FOR

ON/GOSUB выполняет оператор GOSUB в зависимости от значения переменной (многопутное ветвление)

ON/GOTO выполняет оператор GOTO в зависимости от значения переменной (многопутное ветвление)

RETURN заканчивает подпрограмму; возврат к оператору, стоящему после GOSUB

STOP временный останов выполнения с сообщением. Выполнение может быть возобновлено.

1.3.10 Средства обработки ошибок

ERL содержит номер строки, в которой произошла последняя ошибка

ERR содержит номер последней ошибки

ERROR оператор, имитирующий ошибку

ON ERROR GOTO
указывает тип обработки ошибки и адрес программы обработки ошибки

RESUME конец программы обработки ошибки

1.3.11. Средства событийных прерываний

INTERVAL ON/OFF/STOP
определяет прерывание по таймеру

KEY ON/OFF/STOP
определяет прерывания, инициализируемые с клавиатуры

SPRITE ON/OFF/STOP
определяет прерывания, связанные со спрайтами

STOP ON/OFF/STOP
определяет прерывания, связанные с нажатием клавиш Control-STOP

STRIG ON/OFF/STOP
определяет прерывания от триггера игрового пульта

ON event GOSUB
определяет адрес программы обработки прерывания

RETURN осуществляет возврат в основную программу из подпрограммы, обрабатывающей прерывание

1.3.12 Работа с данными, хранящимися в тексте программы

DATA задает список значений данных, хранящихся в тексте программы

READ присваивает значения из оператора DATA переменным

RESTORE восстанавливает указатель для повторного считывания из оператора DATA

1.3.13 Комментарии в программе

REM или ' оператор указывает, что весь последующий текст в строке является примечанием

1.3.14 Доступ к памяти и ввод/вывод

INP ( ) вводит данные из порта ввода

OUT посылает данные в порт вывода

PEEK ( ) читает данные по любому адресу памяти (ОЗУ или ПЗУ)

POKE записывает данные по любому адресу ОЗУ

TIME осуществляет доступ из Бейсика к 16-битному 50 Гц таймеру

USR ( ) обращается к подпрограмме на машинном языке, которую опередилил и за ранее подготовил пользователь

WAIT читает данные из порта ввода и ждет, пока не появится ожидаемое условие.
Примечание: INP, OUT и WAIT используют заранее специфицированные порты. Номера этих портов могут измениться в будущем; по этой причине не рекомендуется употреблять указанные операторы в программах, которые предназначены для тиражирования

1.3.15 Управление экраном дисплея

BASE разрешает поместить в видеопамять (VRAM) таблицы VDP для всех режимов оператора SCREEN

CLOSE останавливает использование файла в/в для вывода на экран

CLS засвечивает экран фоновым цветом

COLOR устанавливает цвет изображения, фоновый цвет, цвет границы (последнее не относится к SCREEN 0)

CSRLIN возвращает номер строки, в которой находится курсор

LOCATE передвигает курсор; включает и выключает курсор

OPEN "CRT:"/"GRP:" открывает файл в/в для вывода на экран текста и графики

POS ( ) возвращает позицию курсора в строке

PRINT # выводит данные на экран в текстовом режиме

PRINT выводит данные в файл на устройстве в/в

SCREEN устанавливает вид изображения и размер спрайта

USING является вспомогательным для оператора PRINT при форматировании вывода

VDP разрешает доступ к регистрам VDP

VPEEK ( ) читает любую ячейку из видеопамяти (VRAM)

VPOKE записывает данные в любую ячейку видеопамяти (VRAM)

WIDTH устанавливает ширину экрана для вывода текста

1.3.16 Графические операторы

CIRCLE рисует окружности, дуги, углы (сектора) и овалы

DRAW использует графический макроязык для изображения объектов

LINE рисует линии, незаполненные прямоугольники, заполненные прямоугольники

PAINT заполняет замкнутые области любым цветом

POINT ( ) возвращает код цвета любой точки экрана

PRESET ( ) делает цвет любой точки экрана совпадающим с фотовым цветом

PSET ( ) делает цвет любой точки экрана совпадающим с цветом изображения

1.3.17 Управление спрайтами

ON SPRITE GOSUB определяет подпрограмму обработки прерывания при "столкновении" спрайта

PUT SPRITE осуществляет управление выводом на дисплей одного (из 32 возможных) спрайта

SPRITE$ ( ) осуществляет доступ к 64 или 256 шаблонам спрайтов

SPRITE ON/OFF/STOP
управляет обработкой прерываний при "столкновении" спрайтов

1.3.18 Управление звуком

BEEP выдает звуковой сигнал и инициализирует устройство вывода звука

PLAY использует музыкальный макроязык для описания музыкальных эффектов (возможно трехголосие)

PLAY ( ) определяет, используется ли буфер оператора PLAY или он пустой

SOUND разрешает доступ к рабочим регистрам PSG (устройства вывода звука)

1.3.19 Ввод/вывод файлов

CLOSE заканчивает в/в файла с заданным номером

EOF ( ) определяет, достигнут ли признак конца файла

INPUT читает элементы данных, которые должны быть в соответствующем формате

INPUT$ ( ) читает любое указанное количество байтов данных

LINEINPUT читает строку текстового файла

OPEN ставит в соответствие имени устройства номер файла; инициализирует файл.

PRINT выводит текст в указанный файл

1.3.20 Ввод с клавиатуры

INKEY$ вводит с клавиатуры строку знаков или пустую строку

INPUT осуществляет ввод с подсказкой; данные должны быть в соответствующем формате

INPUT$ ( ) вводит указанное количество байтов без отображения их на экране

LINE INPUT целиком вводит неформатированную строку

1.3.21 Вывод на печать

LPOS ( ) возвращает текущую позицию головки печатающего устройства

LPRINT посылает данные на печать

OPEN "LPT:" выбирает печатающее устройство печати в качестве файла

1.3.22 Управление кассетным магнитофоном

CLOSE закрывает файл на кассете

MOTOR ON/OFF
запускает и останавливает кассетный магнитофон

OPEN "CAS:"
выбирает кассетный магнитофон в качестве файла в/в

1.3.23. Игровые манипуляторы

PAD ( ) ввод координат с графического планшета

PDL ( ) вводит код положения игрового рычажка

STICK ( ) вводит код положения джойстика

STRIG ( ) вводит сигнал, возникающий при нажатии кнопки на джойстике

1.3.24 Операции

AND/OR/NOT/EQV/IMP/XOR
логические операции

^ , * , / , + , – , \ , MOD
числовые операции

— унарная числовая операция

+ двуместная строковая операция

= , <> , >< , < , > , <= , =< , => , >=

операции отношения; оперируют со строками и числами (результат — целое число, изображающее логическую переменную)

ГЛАВА 2

Общие сведения о языке MSX-Бейсик

MSX-Бейсик имеет много полезных средств. Главное — это то, что иммется много типов переменных, с которыми можно производить различные действия.

Эта глава подробно описывает основные характеристики Бейсика, относящиеся к таким понятиям, как вычисление выражений и хранение информации.

2.1 Режимы работы

После инициализации системы MSX-Бейсик на экран выводится сообщение "OK". "OK" указывает, что MSX-Бейсик находится в режиме команд, т.е. готов воспринимать команды, вводимые с клавиатуры.

С этого момента MSX-Бейсик может быть использован в любом из 2-х режимов: режим прямого, или непосредственного, выполнения команд и режим выполнения программы (RUN).

В прямом режиме MSX-Бейсик перед оператором или командой не ставится номер строки, и выполняются они немедленно после ввода.

Результат арифметических и логических операций может быть выведен на экран немедленно или запомнен для дальнейшего использования.

При этом сами команды после исполнения не запоминаются (теряются).

Прямой режим полезен при отладке и при использовании Бейсика в качестве калькулятора для быстрых вычислений, не требующих сложных программ.

Режим RUN используется для выполнения программ. Строкам программы предшествуют номера строк; строки запоминаются в память.

Запомненная программа выполняется путем ввода команды RUN.

Имеется и третий режим — режим ввода, когда исполняются операторы INPUT или LINE INPUT.

Этот режим похож на первый — прямой, но реализуется только при выполнении указанных операторов.

Более подробная информация о режимах работы приведена в главе 3.

2.2 Формат строки

В MSX-Бейсик строки программы имеют следующий формат (квадратные скобки указывают на необязательную часть текста).

nnnnn оператор Бейсик [:оператор Бейсик...] <нажатие клавиши RETURN>

В строке может быть несколько операторов. При этом каждый должен быть отделен от предыдущего двоеточием. Операторы IF и REM/' должны занимать всю оставшуюся часть физической строки. Подробнее об этом говорится в гл. 3.

Строка программы MSX-Бейсик всегда начинается с номера строки и заканчивается нажатием клавиши RETURN. Строки программы запоминаются в памяти в соответствии со своими номерами.

Номер строки используется как указатель при передачах управления и при редактировании. Номер строки может иметь значение в пределах 0—65529. Для ссылки на номер текущей строки в операторах LIST, AUTO и DELETE вместо номера строки можно использовать точку (.). Текущая строка — это последняя сторока, обработанная текстовым редактором (он встроен в систему MSX-Бейсик), либо та сторока, в которой система обнаружила ошибку.

2.3 Набор символов

В MSX-Бейсик набор символов (знаков) состоит из букв алфвита, цифр, специальных знаков и графических символов.

Буквы в MSX-Бейсик могут быть строчными и прописными. Цифры — в диапазоне от 0 до 9.

Кроме того, в MSX-Бейсик опознаются следующие специальные знаки:

Знак	Действие
	пробел
=	знак равенства или присваивания
+	знак плюс
-	знак минус

*	звездочка или знак умножения
/	знак деления
^	знак возведения в степень
(	левая скобка
)	правая скобка
%	процент
#	номер
$	знак доллара
!	восклицательный знак
[	левая квадратная скобка
]	правая квадратная скобка
,	запятая
.	десятичная точка
"	кавычки
'	апостроф
;	точка с запятой
:	двоеточие
&	амперсанд
?	знак вопроса
<	меньше, чем
>	больше, чем
\	обратная косая черта или деление нацело с остатком
@	знак цены
_	подчеркивание
£	знак фунта
delete	стирание последнего введенного символа на экране
escape	код выхода
tab	табуляция
line feed	переход к следующей строке (перевод строки)
carriage RETURN	заканчивает ввод строки (возврат каретки)

2.4 Константы

Константы являются величинами, используемыми в MSX-Бейсик во время выполнения программ.

Имеются два типа констант: строковые и числовые.

Строковые константы — это последовательность длиной до 255 алфавитно-цифровых знаков, заключенных в кавычки.

Допускается использования знаков кириллицы (прописных и строчных букв).

Примеры:

"HELLO"	"ЗДРАВСТВУЙТЕ"
"$25,000.00"	"Иванов И.И."
"Number of Employees"	"Число учемиков"

Числовые константы являются положительными или отрицательными числами. Числовые константы MSX-Бейсик не могут содержать запятых. Существует 5 типов числовых констант.

1. Целые константы.

Целые числа между –32768 и 32767. Целые константы не могут содержать десятичную точку. Они хранятся в виде целых, двоичных чисел со знаком (16-разрядных). Для некоторых функций от целых чисел вы можете использовать двоичные 16-разрядные числа без знака. В этом случае значения целых чисел в диапазоне от 32768 до 65535 будут соответствовать значениям отрицательных целых чисел в диапазоне от –32768 до –1 (сама ЭВМ не в состоянии отличить 65535 от –1 — это может только программист, прим. пер.). Эти значения, превышающие +32767, невозможно присвоить целым переменным, но их можно использовать в числовых константах (тогда при выполнении команды, LIST они намечаются восклицательным знаком) или в переменных с плавающей запятой. Целые константы без знака, часто используются с определенными функциями, такими, как PEEK ( ).

Примеры:

123
–2345
40100 (**Примечание:** константа без знака, см. выше)

2. Константы фиксированной точкой.

Положительные или отрицательные вещественные числа, т.е. числовые константы, содержащие десятичную точку.

3. Константы с плавающей точкой.

Положительные или отрицательные числа, представленные константами в экспоненциальной форме. Константа с плавающей точкой, возможно со знаком (мантиссы), буквы D или E и целого числа, возможно со знаком, (экспоненты). Допустимый интервал констант с плавающей точкой 10^{64}—10^{-63} (для обозначения экспоненты 10 используются буквы E или D).

Примеры:

```
235.988E-7  =  .0000235988
2359E6      =  2359000000
```

(Для обозначения константы с двойной точностью вместо буквы E используется буква D).

4. Шестнадцатеричные константы.

Шестнадцатеричные числа (по основанию 16) обозначаются префиксом &H. Шестнадцатеричные константы используют цифры от A до F для обозначения величин от 10 до 15. Эти константы могут быть только целыми.

Примеры:

```
&H76
&h32F
```

5. Восмьеричные константы.

Восьмеричные числа обозначаются префиксом &O (O-буква). Восьмеричные (по основанию 8) константы не должны содержать цифры 8 или 9, и могут быть только целыми.

Примеры:

```
&O347
&o177777
```

6. Двоичные константы

Двоичные числа обозначаются префиксом &B. Двоичные (основание 2) числа содержат только цифры 0 и 1 и могут быть только цельми.

Примеры:

```
&B01110110
&b11100111
```

Так как в последних трех олучаях рассматривались только целые константы, следует иметь в виду, что старший бит является знаковым бытом. Для операций без знака это можно не принимать во внимание, но для математических преобразований и при преобразовании выводимых данных любое число с 1 в старшем бите считается отрицательным и представленным в дополнительном коде.

Если необходимо представить число со знаком для математических преобразований, то с отрицательным числом нужно выполнить следующие преобразования: записать абсолютную величину числа, инвертировать каждый бит полученного числа (т.е. заменить все 1 на 0 и все 0 на 1 — прим. пер.), затем добавить 1 к младшему разряду чисел (игнорируя возможный перенос 1 из старшего разряда). Таким образом создается битовое представление для отрицательного числа. Все математические подпрограммы и подпрограммы ввода/вывода выполняют зто автоматически.

2.4.1 Одинарная и двойная точность

Числовые константы могут быть представлены с одинарной и двойной точностью, Числовые константы одинарной точности могут содержать до шести цифр и печатаются с точностью до шести цифр. Числовые константы двойной точности могут содержать до 14 цифр и выдаются на печать с точностью до 14 цифр. По умолчанию для констант в MSX-Бейсик используется двойная точность. Также стандартно округляются лишние цифры по методу 5/4.

Константа с одинарной точностью — это любая числовая константа, состоящая не более чем из 7 цифр и записанная в соответствии с одним из следующих правил:

1. Экспонециальная форма с использованием буквы E.

Или 2. За числом следует восклицательный знак (!).

Примеры:

–1.09E–06
22.5!

Константой двойной точности является любая числовая константа, записанная в одной из следующих форм:

1. Любой набор цифр без экспоненциальной составляющей или указателя типа (по умолчанию число всегда имеет двойную точность)

или 2. Экспоненциальная форма с использованием буквы D

или 3. За числом следует знак #

Примеры:
3489
345692811
–1.09432D–06
3489.0#
7654321.1234

2.5 Переменные

Переменные — это имена, обозначающие области в оперативной памяти, которые Бейсик выделяет для автоматического запоминания величин, используемых в Бейсик — программах. Значение переменной может быть явно задано программистом или получено в результате вычислений в программе. Когда переменная используется впервые, то для ее хранения выделяется область памяти, первоначально заполненная нулями. Объем этой области памяти зависит от типа переменной (или от типа по умолчанию, если тип не задан).

2.5.1 Имена переменных и декларация типов данных.

Имена переменных в MSX-Бейсик могут быть любой длины, но только 2 первых знака являются значимыми; остальные во внимание не принимаются, если только в качестве имени случайно не использовано ключевое слово. Имена переменных могут содержать и буквы, и цифры, но первый знак должен быть буквой. Вожможно также использование специальных символов декларации (см. ниже).

Имя переменной не может быть резервированным словом и не может быть объединением резервированных слов. Резервированные слова включают все коменды, предложения, имена функций и операторов MSX-Бейсик (см. Приложение 1). Если имя переменной начинается с FN, то это обозначает вызов функции, определенной пользователем.

Переменные могут содержать как числа, так и строки. Для обозначения строковой переменной за ее именем ставится знак $. Например, в A$ = "Sales report", $ декларирует тип строковой переменной. Внутреннее представление строк имеет код типа значения, равный 3; это указывает на тот факт, что для каждой строковой переменной или элемента массива помимо имени и типа требуется три байта для запоминания служебной информации (не считая собственно длину строки, на которую указывает переменная).

Имена числовых переменных могут обозначать как целые числа, так и числа с одинарной и двойной точности. Символы декларации типа для имен таких переменных следующие:

символ	название типа	код типа значения
%	целая переменная	2
!	переменная одинарной точности	4
#	переменная двойной точности	8

По умолчанию типом числовой переменной является тип "двойная точность". Код типа значения MSX-Бейсик использует для внутреннего обозначения типа; это количество байтов, необходимое для хранения значения каждой переменной или значения элемента массива указанного типа.

Для хранения имени и кода типа значения каждой переменной требуется дополнительно по 3 байта.

Примеры имен переменных в MSX-Бейсик:

PI#	декларирует величину двойной точности,
MINIMUM!	декларирует величину одинарной точности,
LIMIT%	декларирует величину одинарной точности,
N$	декларирует строковую величину,
ABC	представляет величину двойной точности.

Существует еще один способ декларации типов переменных. Для декларации типов некоторых имен переменных в программу на языке MSX-Бейсик могут быть включены операторы DEFINT, DEFSTR, DEFSNG, DEFDBL (см. более подробное описание в главе 3).

2.5.2 Массивы переменных

Массив — это группа или таблица значений, обозначенных одним именем — именем массива. Каждый элемент массива обозначается именем массива, дополненным целым порядковым номеров — индексом (или целым значением выражения).

Каждый массив переменных имеет количество индексов, равное размерности массива. Например, V (10) обозначает одномерный массив, (1, 4) обозначает двумерный массив и т.д. Максимальная размерность массива-255. Максимальное число элементов массива определяется размером памяти.

Массивы могут быть любого типа. Они более компактны, чем скалярные переменные, поскольку не тратится по 3 дополнительных байта на имя каждой переменной, Массивы также удобно использовать для работы с таблицами данных или для автоматического просмотра набора элементов данных (т.к. индекс может быть переменным).

2.5.3 Требования к памяти

В следующей таблице указано количество байтов, занимаемых переменной типа, указанного в имени переменной.

Тип переменной	Количество байтов	
Целая	5	
Одинарная точность	7	
Двойная точность	11	
Строковая	6	(плюс длина области, занятой самой строкой).

Тип массива	Количество байтов под один элемент	
Целый	2	
Одинарная точность	4	
Двойная точность	8	
Строковая	3	(плюс длина области, занятой самой строкой).

2.6 Преобразование типов

При необходимости система MSX-Бейсик переводит числовые константы из одного типа в другой. Правила преобразования и примеры:

1. Если переменной некоторого типа присваивается значение константы другого типа, число приводится к типу переменной. (Если строковой переменной присваивается числовое значение или наоборот, то фиксируется ошибка: "Type mismatch", "Несоответствие типов").

   Примеры:

```
10 A% = 23.42
20 PRINT A%
RUN
23
OK
```

2. При вычислении выражения все операнды в арифметических операциях и операциях отношения приводятся к одной и той же степени точности — к наиболее высокой степени; результат арифметической операции приводится к исходной степени точности.

Примеры:

```
10 D = 6/7!
20 PRINT D
RUN
 . 85714285714286
OK
```

Арифметическое действие было выполнено с двойной точностью, и полученный результат имеет двойную точность.

```
10 D ! = 6/7
20 PRINT D !
RUN
. 857143
```

Арифметическое действие было выполнено с двойной точностью, результат присваивается переменной с одинарной точностью, поэтому перед присваиванием выполнено округление и напечатана значение переменной с одинарной точностью.

3. Логические операции преобразуют свои операнды к целому виду и возвращают целое значение. Операнды должны быть в диапазоне от -32768 до 32767, в противном случае фиксируется ошибка "Overflow", "Переполнение".

4. При преобразовании числа с плавающей точкой в целое дробная часть отбрасывается.

Пример:

```
10 C% = 55.88
20 PRINT C%
RUN
56
OK
```

Прим. пер.
Из примера видно, что перед отбрасыванием дробной части производится округление

5. Если переменной с двойной точностью присваивается значение с одинарной точностью, то только первые 6 цифр преобразованного числа являются точными. Это происходит потому, что одинарная точность обеспечивает только 6 цифр.

Пример:

```
10 A! = SQR (2)
20 B = A!
30 PRINT A!, B
RUN
1.41421   1.41421
OK
```

Перед отбрасыванием лишних цифр производится округление: анализируется старшая из отбрасываемых цифр и если она $\geqslant 5$, то младшая из оставшихся цифр увеличивается на 1.

2.7 Выражения и операции

Выражение — это строковая или числовая константа, переменная или группа констант и переменных, соединенных между собой знаками операций так, что в результате выполнения этих действий получается единственное значение.

Операции выполняют арифметические и логические действия с операндами. Операции в языке MSX-Бейсик деля тся на 4 типа:

1. Арифметические операции
2. Операции отношения
3. Логические операции
4. Операции-функции

Каждый из типов описан ниже.

2.7.1 Арифметические операции

Арифметические операции в порядке убывания приоритетов:

Операция	Действие	Примеры
^	Возведение в степень	X^Y
—	Отрицательное значение	—X
* , /	Умножение, деление	X*Y
	с плавающей точкой	X/Y
+ , —	Сложение, вычитание	X+Y

Для изменения порядка выполнения действий используются круглые скобки.

Действия в круглых скобках имеют наивысший приоритет. Внутри скобок сохраняется обычный порядок действий.

Деление нацело и вычисление остатка.
В языке MSX-Бейсик имеются две дополнительные операции. Первая — деление нацело (\ ). Перед выполнением деления операнды преобразуются к целому типу путем отбрасывания дробных частей (они должны находиться в интервале от –32768 до 32767), полученное частное преобразуется к целому виду путем отбрасывания дробной части.

Примеры:

```
10\4→2
25.68\5.99→  4
```

Деление нацело по приоритету следует непосредственно за умножением и делением плавающих чисел.

Вторая операция — вычисление остатка обозначается MOD. Результатом вычисления остатка является целая величина, разная остатку от деления целых чисел.

Примеры:

```
10.4 MOD        →   2 (10/4=2 с остатком 2)
25.68 MOD 6.99  →   1 (25/6=4 с остатком 1)
```


По приоритету деление по модулю следует за делением нацело.

Переполнение и деление на 0.
Если при вычислении выражения встретилось деление на 0, то фиксируется ошибка “Division by zero”, “деление на 0”, и выполнение программы прекращается.

При переполнении фиксируется ошибка “Overflow”, “переполнение”, и выполнение программы прекращается.

2.7.2 Операции отношения

Операции отношения применяются для сравнения двух величин. Результатом сравнения будет логическая величина “истина” (1) или “ложь” (0). Результат преобразуется к целому. Результат “истина” возвращает все 1 (что соответствует —1 в дополнительном коде в двоичном представлении). Результат “Ложь” возвращает все 0. Результат операции отношения может быть использован для принятия решения о дальнейшем ходе выполнения программы. Любая величина, отличная от 0, считается результатом “ложь” (см. главу 3, оператор IF).

Операции отношения:

Операция	Проверяемое условие	Примеры
=	равенство	X=Y
< >	неравенство	X< >Y
<	меньше чем	X<Y
>	больше чем	X>Y
<=	меньше либо равно	X<=Y
>=	больше либо равно	X>=Y

(Знак = используется также в операторе LET для присваивания значения переменной.)

При использовании в выражении и арифметических, и логических операций арифметические выполняются первыми.

Например, выражение

```
X+Y < (T-1) / Z
```

истино, если величина X+Y меньше, чем величина (T—1) /Z

Еще примеры:

```
IF SIN (X) < 0 GOTO 1000
IF I MOD J+0 THEN K=K+1
```

2.7.3 Логические операции

Логические операции — поразрядные операции. Они выполняются над каждой парой соответствующих битов. Результатом логической операции является TRUE ("Истина", ≠0) или FALSE ("Ложь", 0). По приоритету логические операции следуют за арифметическими операциями и операциями отношения. Логические операции выполняются по правилам, приведенным в таблице 1. Операции в таблице перечислены в порядке убывания приоритетов.

Таблица 1.

NOT

X	NOT X
1	0
0	1

AND (И, конъюнкция)

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR (ИЛИ, дизъюкнция)

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR (исключающее или)

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

EQV (эквивалентность)

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

IMP (импликация)

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

Так же, как и операции отношения, логические операции могут быть использованы для принятия решения о дальнейшем ходе выполнения программы.

Примеры.

```
IF D=200 AND F<4 THEN 80
IF I=10 OR K<0 THEN 50
IF NOT P THEN 100
```

Логические операции выполняются над 16-битными целыми операндами со знаком которые должны принадлежать интервалу -32768; +32767, в противном случае фиксируется ошибка. Если оба операнда являются 0 или оба –1, то в результате логической операции будет получен либо 0 либо –1. Логическая операция выполняется над целыми побитно, т.е. каждый бит результата определяется значениями соответствующих битов в операндах.

Таким образом, можно применять логические операции для анализа содержимого байта по маске. Например, операция AND может быть использована для маскирования всех битов, кроме одного, в байте состояния порта ввода/вывода ЭВМ.

Операция OR может быть использована для "слияния" двух байтов. Следующие примеры демонстрируют действие логических операций.

```
63 AND 16 → 16
        63 = двоичное 111111
        16 = двоичное 10000

15 AND 14 → 14
        15 = двоичное 1111
        14 = двоичное 1110
```

-1 AND 8 → 8

−1 = двоичное 1111111111111111

8 = двоичное 1000

4 OR 2 → 6

4 = 100,

2 = 10, и таким образом 110 = 6

10 OR 10 → 10

10 = двоичное 1010.

-1 OR -2 → -1

-1 = 1111111111111111

-2 = 1111111111111110

(обратный код к 16-ти нулям есть 16 единиц, которые соответствуют представлению -1 в дополнительном коде).

NOT X → – (X+1)

Дополнительный код числа в двоичной системе получается добавлением 1 к обратному коду числа.

2.7.4 Операции — функции

Функция используется в выражении для выполнения опеределенного действия над операндаом. Язык MSX-Бейсик содержит такие встроенные функции, как SOR ( ) (корень квадратный) или SIN ( ) (синус).

MSX-Бейсик допускает использование функций, определенных программистом (см. описание DEF FN в главе 3).

2.7.5 Строковые операции

Строки могут объединяться с помощью знака "+".

Пример:

```
10 A$=$="FILE":B$="NAME"
20 PRINT A$+B$
30 PRINT "NEW"+A$+B$
```



```
RUN
FILENAME
NEW FILENAME
OK
```

Строковые переменные могут сравниваться с помощью тех же операций отношения, что и числовые:

= <> < > <= >=

Строки сравниваются посимвольно. Значением символа является его код по таблице кодов ASCII кодавая таблица расширена для КУВТ ЯМАХА путем включения знаков кириллицы — прим, ред. Если все коды совпадают, то строки равны. Если коды не совпадают, то меньшее значение кода предшествует большему. Если строки имеют разную длину, и при сравнении достигнут конец более короткой, то короткая строка считается меньшей. Пробелы в начале и в конце строки значимы. Примеры (все имеют значение "TRUE" — "истина")

```
"AA" < > "AB"
"FILENAME" = "FILENAME"
"X&" > "X"
"CL" > "CL"
"kg" > "KG"
"SMYTH" < "SMYTME"
B$ < "9/12/83" (где B$ = "8/12/83")
```

Таким образом, строковые операции могут быть использованы для проверки строковых величин и расположения строк в алфавитном порядке. Все строковые константы, используемые в выражениях, должны быть заключены в кавычки.

2.8 Редактирование программ

Экранный редактор позволяет пользователю вводить строки программы обычным путем, а затем редактировать полный экран до внесения изменений. Эта ускоряющая работу возможность обеспечивается с помощью специальных функциональных клавиш, управляющих перемещением курсора, режимом вставки или удаления знаков, стирания строки или экрана, а также с

помощью некоторых сочетаний нажатия клавиши Control с буквенными клавишами. Конкретные функции и их задание описываются ниже.

При помощи экранного редактора пользователь может быстро передвигать курсор по экрану, внося при необходимости исправления. Исправления вносятся при помощи перемещения курсора на первую строку, подлежащую изменению, и нажатия клавиши RETURN в начале каждой строки. Строка программы не изменяется до тех пор, пока не будет нажата клавиша RETURN при нахождении курсора в этой строке.

2.8.1 Создание программ

Программы создаются при помощи экранного редактора. В системе MSX-Бейсик можно редактировать программу сколь угодно долго после подсказки OK и до отдачи приказа RUN (прямое управление). Каждая вводимая строка текста обрабатывается редактором. Если строка текста начинается с номера, то эта строка является оператором программы.

Операторы программы обрабатываются редактором одним из следующих способов:

1. К программе должна быть добавлена новая строка. Это происходит, если значение номера строки заключено в интервале [0:65529] и за номером строки следует хотя бы один знак, отличный от пробела.

2. Строка должна быть изменена. Это происходит, если номер строки совпадает с уже имеющимся в программе номером строки. Старая строка заменяется на новую.

3. Удаление строки происходит, если номер строки совпадает с номером имеющейся строки и новая строка содержит только номер.

4. Допущена ошибка.

Если сделана попытка удалить несуществующую строку, фиксируется ошибка “Underfined lline number”, “неопределенный номер строки”.

Если память, отведенная для программы, исчерпана и должна быть добавлена строка, фиксируется ошибка “Out of memory”, “переполнение памяти”, и новая строка не добавляется.

В строке может размещаться более чем 1 оператор. В этом случае операторы разделяются знаком (:). Ни перед, ни после (:) пробелы, могут не ставиться.

Максимальное число знаков в строке программы, включая номер строки, равно 255.

2.8.2 Редактирование программы

Чтобы отредактировать программу, необходимо поместить ее или часть ее строк на экран с помощью оператора LIST.
Текст изменяется перемещением курсора в нужное место и выполнением одного из следующих действий:

1. Заменой существующего знака (перепечатка)
2. Удалением знака справа от курсора
3. Удалением знака слева от курсора
4. Вставкой знаков
5. Добавлением знаков и конец логической строки (т.е. строки программы, а не экрана.-прим. пер.).

Эти действия выполняются при помощи специальных функциональных клавиш, реализующих различные функции экранного редактора (см. следующий раздел).

Изменения строки производятся при нажатии клавиши “возврат каретки” в то время, как курсор находится где-либо в строке. Нажатием клавиши RETURN вводятся все изменения в логическую строку независимо от количества физических строк и независимо от положения курсора в строке.
Заметьте, что каждый раз, когда вы вводите символ, который переносит вас из конца одной строки в начало другой (не управляя курсором), эти две строки считаются одной логической стро-

кой. Если вы работали в режиме вставки при этом курсор имеет половинную высоту), то когда вы вносите знак в конец строки, все остальные (все последующие физические строки) смещаются вниз и для продолжения логической строки вносится пустая строка.

2.8.3 Функции полноэкранного редактора

В следующей таблице приводятся шестнадцатеричные коды для управляющих функций системы MSX-Бейсик и указываются эти функции. Приводятся также последовательности, полученные при нажатии клавиши Control и другой клавиши, соответствующие этим функциям. Они максимально соответствуют стандарту ASCII. Некоторые управляющие функции описаны вслед за таблицей более подробно.

Управляющие функции системы MSX-Бейсик

Код клавиши Control, часто обозначаемый с последующей буквой, — например, C вводится путем нажатия клавиши Control и, не отпуская ее, клавиши соответствующей буквы. Ниже приведен список кодов управления при нажатой клавише Control:

Шестнадцатеричный код	Управляющая клавиша нажата	Специальная клавиша	Функция
01	A		Заголовок графического символа
*02	B		Перемещение курсора к началу предыдущего слова
*03	C		Прерывание, когда система MSX-Бейсик ожидает ввод
*04	D		Игнорируется
*05	E		Усечение строки (очистка до конца логической строки)

*06	F		Перемещение курсора к началу следующего слова
*07	G		Звуковой сигнал
08	H	Back Space	Стирание последнего введенного символа
09	I	TAB	Табуляция (8 знаков)
*0A	J		Перевод строки
*0B	K	HOME	Установка курсора в начальное положение (левый верхний угол экрана)
*0C	L	CLS	Очистка экрана
*0D	M	RETURN,	Возврат каретки, ввод логической строки
*0E	N		Дополнение в конец строки
*0F	O		Игнорируется
*10	P		Игнорируется
*11	Q		Игнорируется
*12	R	INS	Режим вставки/замены с триггерным переключением
*13	S		Игнорируется
*14	T		Игнорируется
*15	U		Удаление логической строки
*16	V		Игнорируется
*17	W		Игнорируется
*18	X	SELECT	Игнорируется
*19	Y		Игнорируется
*1A	Z		Игнорируется
*1B	[	ESC	Игнорируется
*1C	\	→	Курсор вправо
*1D	]	←	Курсор влево

*1E	^	↑	Курсор вверх
*1F	—	↓	Курсор вниз
7F	Delete	DEL	Удаление символа, указанного курсором. Текст сдвигается влево

(Коды, отмеченные *, выводят из режима вставки, если вы работали в этом режиме) .

Предыдущее слово:
Курсор смещается влево к предыдущему слову. Предыдущим словом считается знак слева от курсора и принадлежащий одному из следующих множеств: A—Z; А—Я; a—z; а—я; 0—9.

Прерывание:
Возвращает MSX-Бейсик в режим прямого управления при нажатии клавиши RETURN, без сохранения изменений, внесенных в редактируемую логическую строку.

Усечение:
Все символы логической строки, начиная от текущего положения курсора до конца логической строки удаляются.

Следующее слово:
Курсор передвигается вправо к следующему слову. Следующее слово — знак справа от курсора, принадлежащий одному из множеств A—Z; А—Я; a—z; а—я; 0—9.

Звуковой сигнал:
Слышен звуковой сигнал, такой же, как и при выполнении оператора BEEP

Стирание последнего символа:
Уничтожается знак слева от курсора.
Все знаки справа от курсора перемещаются влево на одну позицию.
Последующие знаки и строки в логической строке передвигаются вверх.

Табуляция:
В режиме вставки функция TAB вводит пробелы с обозначенной курсором позиции до следующей позиции табуляции.

Происходит раздвигание строк с переходом на следующую строку.
В режиме, отличном от режима вставки, функция TAB перемещает курсор к следующей позиции табуляции. Позиция табуляции находится через каждые 8 позиций.

Перевод курсора в исходное положение:
Курсор перемещается в левый верхний угол экрана. Экран не очищается.

Очистка экрана:
При нажатии этой клавиши курсор переходит в исходное положение, и очищается весь экран, независимо от исходной позиции курсора.

Возврат каретки:
Вставляет код "возврат каретки" в текущей позиций курсора. Нажатие клавиши "возврат каретки" указывает система MSX-Бейсик на логический конец данной строки. Курсор перемещается в начало следующей логической строки; если на экране недостаточно места, то все изображение сдвигается вверх и снизу добавляется пустая строка.

Дополнение в конец строки:
Переводит курсор в конец строки, остающиеся знаки в строке не стираются. С новой позиции все внесенные знаки добавляются к логической строке, пока не произойдет нажатие клавиши "возврат каретки".

Вставка:
Триггерный переключатель для режима вставки. При режиме вставки размер курсора уменьшается, и начиная с данной позиции происходит вставка знаков. При вставке новых знаков знаки справа от курсора перемещаются вправо. Соблюдается непрерывный переход на новые строки.
По окончании режима вставки размер курсора вновь увеличивается и внесенные знаки заменяют знаки в строке.

Очистка логической строки:
При нажатии этой клавиши в любой части строки стирается вся логическая строка.

Курсор вправо:
Курсор перемещается вправо на одну позицию. Соблюдается непрерывные переход на последующие строки.

Курсор влево:
Курсор переводится влево. Соблюдается непрерывный переход на предыдущие строки.

Курсор вверх:
Курсор переводится вверх на одну физическую строку (в данной колонке).

Курсор вниз:
Курсор перемещается на одну физическую строку вниз (в данной колонке).

2.8.4 Определение логической строки для оператора INPUT

Логическая строка может состоять из нескольких физических строк (т.е. строк экрана — прим. пер.). Однако при выполнении операторов INPUT и LINE INPUT это определение несколько изменяется. При выполнении любого из упомянутых операторов в состав логической строки входят только введенные с клавиатуры символы, или символы, через позиции которых был перемещен курсор.

Перемещение символов, входящих в состав логической строки, вызывают только вставка и стирание. Нажатие клавиши DELETE уменьшает размер строки.

В режиме вставки логическая строка увеличивается, за исключением тех случаев, когда перемещаемые знаки вносятся на место отличных от пробелов знаков, входящих в ту же физическую строку, но не входящих в логическую строку.

В этом случае, знаки, отличные от пробелов, не входящие в логическую строку, сохраняются, а знаки в конце логической строки отбрасываются.

Это сохраняет метки, введенные до выполнения оператора INPUT. Если во время внесения строки был введен неверный знак, он уничтожается при помощи клавиши Backspace или клавиши Control-H. После уничтожения знака продолжайте ввод строки.

Чтобы целиком стереть строку в процессе ее ввода, нажмите клавишу Control-U. Если требуется исправить строку в программе, находящейся в данный момент в памяти, просто заново наберите ее с тем же номером. Система MSX-Бейсик автоматически заменит строку новой.

Чтобы уничтожить всю хранящуюся в памяти программу, введите команду NEW. Команда NEW всегда вводится для очистки памяти перед вводом новой программы.

2.8.5 Одновременное нажатие управляющих клавиш

Если ячейка памяти ENSTOP (адрес FBBO) имеет ненулевое значение (введенное с помощью POKE), то одновременное нажатие клавиш Control, Shift, Graph и Code-Lock вернет систему MSX-Бейсик в прямой режим и на экран будет выведена подсказка OK.

2.9 Специальные клавиши

Система MSX-Бейсик обладает рядом возможностей, обеспечиваемых одновременным нажатием ряда клавиш. Подробно см. приложение J (Таблица). MSX, однако, имеет дополнительные специальные управляющие клавиши, описанные ниже.

2.9.1 Функциональные клавиши

MSX-Бейсик имеет 10 функциональных клавиш. Их значения выводятся в последней строке экрана (можно вывести максимально возможное количество символов, если не действует KEY OFF). При помощи оператора KEY значения этих клавиш могут быть переопределены. Изначально опеределены следующие значения функциональных клавиш:

F1 color[b]	[b] = пробел
F2 auto[b]	[cr] = возврат каретки
F4 goto[b]	[u] = курсор вверх
F5 list[b]	[cls] = очистка экрана
F5 run [cr]	

F6 color 15,4,4[cr]
F7 cload"
F8 cont[cr]
F9 list.[cr] [u] [u]
F10 [cls] run [cr]

(При работе в локальной сети КУВТ Ямаха действует другое назначение функциональных клавиш, см. соответствующее описание — прим. ред.)

Функциональные клавиши могут использоваться как клавиши прерывания по событию. См. подробно операторы ON KEY, GOSUB, KEY ON/OFF/STOP.

2.9.2 Клавиша STOP

При использовании экранного редактора нажатие только клавиши STOP не влияет на его работу.

Когда система MSX-Бейсик выполняет программу (режим RUN), нажатие клавиши STOP вызывает подавление выполнения программы, и MSX-Бейсик выводит на экран курсор, чтобы показать, что программа приостановлена. При повторном нажатии клавиши выполнение возобновляется. При одновременном нажатии клавиши Control и клавиши STOP (Control-STOP), система MSX-Бейсик прекращает выполнение программы и возвращается к командному режиму с выводом сообщения "Break in nnnn"; nnnn обозначает тот номер строки программы, где было остановлено выполнение. При этом выводится звуковой сигнал. Вы можете предотвратить действие одновременного нажатия клавиш Control-Stop, если в программе используются операторы ON STOP GOBUS и STOP ON; паузу при нажатии клавиши STOP предотвратить нельзя (заметьте, что INPUT$ ( ) временно отменяет ее).

2.10 Сообщения об ошибках

Если в результате ошибки выполнение программы прекращается, то на экран выводится сообщение об ошибке. Смотри в Приложении полный список кодов ошибок, фиксируемых системой MSX-Бейсик.

ГЛАВА 3

Алфавитный указатель ключевых слов языка MSX-Бейсик

Режимы работы

Система MSX-Бейсик обеспечивает ряд режимов работы: режим выполнение программы, прямой режим и режим ввода.

В прямом режиме программа не выполняется. Можно вводить дюбую логическую строку (она может состоять из нескольких физических строк и содержать не более 255 символов); ввод логической строки заканчивается, если нажата клавиша RETURN. Система анализирует строку и выполняет указанное в строке действие. Если строка начинается с допустимого порядкового номера, то текущая программа изменяется; в противном случае (приказ) немедленно выполняется действие, указанное в строке. Если вы видите системную подсказку Ok, то знайте, что система находится в прямом режиме.

Режим RUN (также называемый косвенным режимом, или режимом выполнения) переводит систему в режим выполнения программы. В режим выполнения можно перейти, либо задав команду RUN в прямом режиме, либо после загрузки в память программы на языке Бейсик или на машинном языке с параметром "R", а также при вставке в разъем расширителя магистрали кассеты, содержащей программу, записанную в ПЗУ. В этом режиме, если программа написана на Бейсике, каждая строка проверяется в очередности номеров и выполняется. Подсказка OK появляется в режиме RUN в следующих случаях: — когда выполнена последняя строка (с максимальным номером) Бейсик — программы;
—при выполнении оператора STOP или END; — при нажатии комбинации клавиш Control-STOP,
а также при обнаружении не перехваченной ошибки. Во всех этих случаях система переходит в прямой режим.

В режим ввода система переходит в двух случаях:
— при выполнении команд INPUT и LINE INPUT (прямой режим)
— при выполнении операторов INPUT и LINE INPUT (режим RUN)
В этом случае вы можете ввести в машину данные, которые будут использоваться вашей программой. При нажатии клавиши RETURN происходит возврат к прерванному месту программы; при обнаружении не перехваченной ошибки система переходит в прямой режим.

Обратите внимание на следующее: такие клавиши как RETURN, Control-C, Control-STOP соответствуют стандартной клавиатуре MSX. В большинстве случаев клавиша Control — это клавиша смены регистра, т.е. вы должны нажать клавишу Control и не отпуская ее, нажать нужную клавишу. Кроме того, система MSX обладает возможностью автоматического повтора. Это означает, что нет необходимости нажимать требуемую клавишу много раз; если вы нажали клавишу, то последний введенный символ после некоторой задержки многократно повторяется системой.

Типы ключевых слов

Ниже в алфавитном порядке приводится описание команд, операторов, функций и псевдопеременных. Каждый раздел описания имеет определенный формат (см. ниже).

В правом верхнем углу страницы указывается тип ключевого слова, ниже дается объяснение.

Команды — ключевые слова, используемые для изменения режимов работы ЭВМ. Они вводятся независимо друг от друга.

Примеры: RUN, LIST, NEW

Операторы — ключевые слова, которые состовляют словарь языка Бейсик. Они последовательно выполняются в ходе работы программы. Примеры: PRINT, INPUT и OPEN.

Команды обычно вводятся в прямом режиме; операторы обычно используются в строках программы. Однако, в большинстве случаев это разделение ролей может быть нарушена. Каждая строка может содержать либо команду, либо оператор, либо несколько операторов, разделенных. " : ".

Функции используются в выражениях. Им передаются аргументы, которые используются при выполнении определенных для данной функции действий. Значение результата присваивается имени функции в выражении. В этом смысле функции аналогичны константам или переменным. Имена некоторых функций: ASC ( ), CHR$ ( ), POS ( ).

Псевдопеременные — особый случай функции. Они могут быть вычислены или им могут быть присвоены значения, определяющие некоторые специальные случаи использования системы. Часто они непосредственно связаны с одной или несколькими переменными в ОЗУ, хотя иногда при обращении к ним выполняются определенные операции — особенно, когда псевдопеременная находится слева в операторе присваивания (LET). В целом функции (за исключением MID$) используются только в выражениях или в правой части оператора присваивания; псевдопеременные используются слева в оператора присваивания или там, где может быть использована обычная переменная. Примеры псевдопеременных: BASE ( ), MAXFILES =, SPRITE$ ( ), VDP ( ).

Системные переменные RAM находятся в ОЗУ и используются системой MSX-Бейсик. Эти участки памяти находятся в верхней части адресного пространства памяти и активно используются на всех этапах работы системы. В приложении приводится распределение памяти для этих переменных. Кроме того иногда приходится обращаться к точкам входа в исполнительную систему BIOS, находящуюся в ПЗУ. В начале памяти находится набор точек входа в ПЗУ системы MSX-Бейсик для выполнения функций ввода — вывода. Список этих точек приводится в приложении.

В микропроцессоре Z-80 слово-это 2 байта (16 бит); оно хранится в обратном порядке (первым записывается младший байт).

Формат справочного материала

Каждое ключевое слово описано по следующим правилам:

Все адреса в скобках являются шестнадцатеричными. Знак :: = означает — определяется как.

Курсив, квадратные и фигурные скобки и многоточие используются как метасимволы — они не предназначены для ввода; они определяют параметры, необязательные параметры и альтернативные параметры. Пример в стандартной нотации:

[текст] — в опереторе текст является необязательным.

{опция 1 / опция 2} — выбор одного из вариантов

идентификатор — здесь слово "идентификатор" описывает параметр, вводимый пользователем

. . . — предыдущий элемент может повторяться многократно.

Слова, написанные заглавными буквами, являются ключевыми; они должны вводиться точно так, как написаны. Прочие слова, написанные заглавными буквами, за которыми следует шестнадцатеричное четырехзначное число, заключенное в скобки, обозначают системные переменные (см. приложения). Далее приводится подробное описание этого формата.

Цель: Здесь дается краткое определение значения ключевого слова

Формат: Ключевое слово *обязательный параметр 1* [, *необязательный параметр 2*]

Обязательный параметр 1 :: = определение и допустимое значение
Необязательный параметр 2 :: = определение и допустимое значение

Действие: Точно определяется назначение ключевого слова, все возможные результаты и параметры.

Дается подробное объяснение, достаточное для программистов, использующих машинный язык, со ссылками на входные точки BIOS — такие, как WRTVRM (00D4) и системные переменные — такие, как VMODE (F73E).

Использование: Здесь поясняются интересные возможности использования данного ключевого слова.

Пример:

```
PRINT "this is a test"
this is a test
Ok
```

См. также: Упоминаемые или аналогичные по действию ключевые слова и разделы настоящего руководства.

Цель: Возвращает значение, равное абсолютной величине аргумента

Формат: *x* = ABS (*арифметическое выражение*)

Арифметическое выражение :: = любая численная величина или выражение

Действие: Эта функция присваивает аргументу знак плюс и возвращает полученное значение.

Использование: ABS ( ) полезна при вычислении и сравнении двух различных чисел.

Например: IF ABS (a–b) > ABS (c–d) THEN . . .

будет оценено правильно независимо от того больше ли a и c чем b и d.

Пример:

```
a# = -1/3: PRINT ABS (a# * 2) + 2
2.6666666666667
```

См. также: INT ( ), SGN ( )

Цель: Преобразование одного байта из строкового формата в числовой

Формат: *x* = ASC (*строковое выражение*)

Строковое выражение :: = любое строковое выражение

Действие: Эта функция выбирает и анализирует первый байт строкового выражения (левый байт). Значение этого байта есть целое число, соответствующее коду символа, в этом байте
0 = < результат < = 255

Если строковое выражение не содержит стмволов (пустое), то фиксируется ошибка "Illegal function call", "Неверное обращение к функции".

Использование: ASC ( ) и обратная ей CHR$ ( ) используются для хранения и извлечения компактно располагаемых небольших по значению величин, Если необходимо проанализировать длинную строку, то можно использовать ASC и MID$:

```
X = ASC (MID$ (Z$, 2))
```

Нет необходимости указывать длину поля, т.к. ASC работает с одним байтом. Помимо того, при вводе символа с помощью INPUT$ (1) ASC ( ) используется для преобразования символа строки в число в соответствии с кодом ASCII (вариант кувт ямаха), что очень полезно:

```
ON ASC (INPUT$ (1)) – 31 GOTO 100, 100, 220, . . .
```

Пример:

```
ZZ$ = "ABC" : PRINT ASC (ZZ$)
65
Ok
```

См. также: CHR$ ( ), STRING$ ( )

Цель: вычисление arctg; результат — функция в радианах

Формат: *x* = ATN (*арифметическое выражение*)

арифметическое выражение :: = любое числовое выражение

Действие: Эта функция вычисляет угол, тангенс которого равен заданному аргументу. Заметьте, что результат выдается в радианах, поэтому он принадлежит интервалу $\left[\frac{\pi}{2} \text{ и } \frac{\pi}{2}\right]$

Выражение x = TAN (ATN (x)) — тождество. Эта функция — обратная функции TAN ( );

TAN ( ) не определен для $-\frac{\pi}{2}$ и $\frac{\pi}{2}$, и для

углов, близких к $-\frac{\pi}{2}$ и $\frac{\pi}{2}$ при вычислении

ATN ( ) может возникнуть потеря точности, т.к. значения тангенса либо очень малы, либо очень велики.

Использование: ATN ( ) — единственная обратная тригонометрическая функция, которая может быть использована; поэтому она незаменима при любых вычислениях с углами или с комплексными числами.
Результат может быть представлен в виде:
Градусы = ATN (arg) * 180/ π
См. в приложении а производные функции и краткое изложение их тригонометрических свойств.

Пример:

```
PRINT ATN (9.9999999999999 e 62) * 2    (максимальное допустимое число)
PRINT ATN (1) * 4    (те же результаты, что и выше)
3.1415926535898    (π с 14 знаками)
Ok
```

См. также: SIN ( ), COS ( ), TAN ( ), приложение А "Производные функции".

Цель: Обеспечивает автоматическую нумерацию строк для редактирования программы.

Формат: AUTO [*начало*] [, *приращение*]

начало :: = числовая константа от 0 до 65529 или точка
приращение :: = числовая константа от 1 до 65529 (по умолчанию значение и начала, и приращения равно 10). Точка (.) обозначает номер текущей строки.

Действие: Эта команда переводит редактор из прямого режима в автоматический.
В результате после каждого нажатия клавиши RETURN следующая строка автоматически получает больший номер. Курсор устанавливается после номера строки и пробела. Вы можете ввести текст строки (если этот текст начинается с числа, то он присоединяется справа к номеру строки). Если номер строки совпадает с уже существующим, то сразу за номером строки появляется [*] Если вы не хотите корректировать имеющуюся строку, нажмите клавишу RETURN, не вводя текст.

Для выхода из автоматического режима нумерации нужно нажать клавиши Control-STOP или Control-C; AUTO возвращается в прямой режим.

Первый аргумент всегда указывает номер начальной строки, второй — приращение между номерами. По умолчанию оба аргумента имеют значение 10. Если указан только первый аргумент и стоит " , ", то величина приращения сохраняет предыдущее значение. Однако, если стоит " , ", но не указан 1-й аргумент, то 1-й аргумент устанавливается в 0. Точка используется на месте первого аргумента для ссылки на текуций номер строки.

Использование: Команда AUTO используется в трех случаях:
— для экономии времени при вводе программы с клавиатуры, т.к. ручную нумерацию строк можно опустить;
— для ускорения ввода повторяющихся задач — таких, как, например, ввод 50 строк данных, особенно, если вы заранее присвоили какому-либо функциональному ключу текст "DATA"; и, наконец,
— для ввода строк программы во время выполнения.
Заметьте, что когда номер строки превышает 65529, этот режим автоматически прерывается и осуществляется переход к прямому режиму.

Пример:

```
110 END
AUTO 100
100 PRINT "I typed this part"
100* END  (здесь вы нажимаете RETURN)
120       (здесь вы нажимаете Control-C)
LIST
100 PRINT "I typed this part"
110 END
Ok
```

См. также: Раздел о формате номера строки.

Цель: Анализ и установка организации видео-памяти в любом режиме

Фармат: BASE (*табл. входов*) = указатель видео-памяти
или
x = BASE (*табл. входов*)
Таблица входов :: = любое числовое выражение, значение которого заключено между 0 и 19.
Указатель лидео-памяти :: = любое числовое выражение, зачение которого является допустимым адресом для видео-памяти (обчно между 0 и 14336).

Действие: BASE ( ) используется для запроса и определения местоположения таблиц видео-памяти (VDP RAM) Двадцать возможных параметров соответствуют двадцати системным переменным (каждая по 2 байта) начиная с адреса (F3B3). Эти параметры зазбиты на 4 группы.
Каждая из этих четырех групп, содержащих по пять параметров, при входе определяет установку VDP RAM, соответственно для режимов SCREEN 0, SCREEN 1, SCREEN 2, SCREEN 3. Более того, когда BASE находится в левой части оператора присваивания, а параметры соответствуют тукущему режиму SCREEN, то регистры VDP корректируются немедленно. В остальных случаях BASE только задает и читает значения системных переменных.
При задании оператора SCREEN x эти переменные задают значения в регистрах VDP (это можно сделать и вручную, используя псевдопеременную VDP). (если, однако, BASE ( ) находится в левой части оператора присваивания и значение, вычисляемое по выражению, лишено смысла для регистра VDP, заданного параметром, то фиксируется ошибка "Illegal function", "неверная функция".
Пятью параметрами для каждого режима SCREEN являются:

0: База таблицы Имен
1. База таблицы Цветов (не используется в SCREEN 0 и 3)
2. База таблицы Генератора шаблонов
3. База таблицы Атрибутов Спрайтов (не используется в SCREEN 0)
4. База таблицы Шаблонов Спрайтов (не используется в SCREEN 0) Таким образом, параметр для BASE определяется следующим путем:

BASE (SCREEN режим * 5 + изменяемый адрес базы)

для нахождения соответствующего места в ОЗУ можно использовать:

&HF3B3 + 2 * (SCREEN режим * 5 + номер изменяемого адрес базы).

BASE ( ), в отличие от SCREEN, не выполняет инициализацию режима Экрана.

Использование: **Заметьте:** использование команды BASE требует знания работы микросхемы VDP. Смотри подробно Справочное руководство TMS—9918 компании Texas Instruments.

В ограниченных масштабах (если вы не используете какой-либо из режимов SCREEN), BASE обеспечивает возможность обращения к функциям машинного языка. BASE также избавляет вас от вычислений, необходимых для преобразования нужных адресов BASE ( ) в требуемые значения и помещения их в регистры VDP.

Вообще говоря, эта псевдопеременная позволяет одновременно задавать в VDP RAM содержимое нескольких различных кадров дисплея, возможно даже в нескольких режимах, по желанию переходя от одного к другому. Смотри подробно

главу, в которой описываются графические возможности.

Программами BIOS, находящимися в ПЗУ и изменяющими режим экрана без его очистки, являются: SETTXT (0078), SETT32 (007B), SETGRP (007E) и SETMLT (0081) для SCREEN 0, 1, 2, 3.
При этом считается, что вы используете BASE для установки адресов таблиц; они не меняют положение курсора, оставляя его в положени, заданном для последнего режима SCREEN.
Вы можете получить доступ к этим входным точкам, используя:

DEF USR0 = &H0078 : фиктивное выражение ,
USR 0 (фиктивное выражение)

Эта процедура не корректирует системные переменные NBMBAS (F922), CSRY (F3DC), CSRX (F3DD) или LINLEN (F3B0) текущее имя таблицы и ширину экрана. Эти значения опеределяют положение курсора при выводе текста на экран дисплея. (Конечно, вы можете, используя один кадр, напечатать — PRINT · – текст в другом кадре) .

Например:

(значения даются по умолчанию)

```
FOR MD = 0 TO 3: FOR TB = 0 TO 4: PRINT BASE (TB + 5*MD);: NEXT
TB: PRINT: NEXT MD
0   0   2048   0   0                    (SCREEN 0)
6144   8192   0   6912   14336          (SCREEN 1)
6144   8192   0   6912   14336          (SCREEN 2)
2048   0   0   6912   14336             (SCREEN 3)
```

(Этот пример указывает адреса таблиц по умолчанию, устанавливаемые для VDP: Имя, Цвет, Шаблон, Атрибут Спрайта и Шаблон Спрайта.)
Для справок приводим список максимальных длин для каждой из тоблиц данного примера. Формат тот же самый.

960, n/a, 2048, n/a, n/a (n/a – лишнего смысла)
768, 32, 2048, 128, 2048
768, 6144, 6144, 128, 2048
768, 2048, n/a, 128, 2048

Запомните, что во всех случаях по умолчанию VRAM, никогда не используется в интервале адресов от 7040 до 8191.

Смотри также: VDP ( ), VPEEK ( ), VPOKE, SCREEN. раздел о графическом программировании.

Цель: Выдает звуковой сигнал и инициализирует устройство вывода звука.

Формат: BEEP параметров нет.

Действие: Этот оператор используется для вывода звукового сигнала. После выполнения BEEP регистры звуковой микросхемы и параметры оператора PLAY по умолчанию устанавливаются в первоначальное состояние и очередь на выбор данных оператором PLAY прерывается. Когда система обнаруживает ошибку, выполняется оператор BEEP.

Оператор BEEP можно эмулировать с помощью оператора PLAY:

PLAY "M255 V8 L64 O6 T120 S1 e"
(L4 O4 — по умолчанию)
или просто
PLAY "O6e64"
(по умолчанию)

Суть оператора BEEP видна из следующего примера:

```
10 SOUND 0, &O125. SOUND 1,0: SOUND 7,
&B10111110
20 SOUND 7, &B10111111
30 SOUND 8, 7
40 x = LOG (x) ' это вызывает задержку в .025
скунд.
```

Использование: Обычно используется для сигнализации о наличии ошибок, обнаруживаемых программой пользователя. Действие этого оператора аналогично выводу кода (07), Control-G на экран (ASCII BEL) и может быть эмулирован командой PLAY, чтобы не вызывать сброса звуковых регистров. Любая ошибка, допущенная в программе на языке MSX — Бейсик, влечет за собой

выполнение BEEP с одновременным сообщением об ошибке.
Таким образом, в случае не перехваченной ошибки выполняется BEEP; поэтому наиболее быстрым способом приведения звуковых регистров в исходное положение при прямом режиме является ввод неправильной команды — Например, P <возврат каретки>.

Пример:

```
125 INPUT "month number"; MN
130 IF MN < 12 OR MN < 1 THEN BEEP: GOTO 125
140 PRINT MONTH$ (MN)
```

См. также: PLAY, SOUND, ERROR

Цель: Преобразование целого числа в двоичный символьный формат.

Формат: *x$* = BIN$ (*Целое выражение*)

Целое выражение :: = любое арифметическое выражение, которое можно вычислить как целое.

Действие: Эта функция вычисляет выражение. Результат преобразуется в целое со знаком или без знака; если это число умещается в пределах — 32768 =< целое выражение < = 65535, фиксируется ошибка переполнения. "Overflow". Полученное целое число преобразуется в строку символов, соответствующую его двоичному коду. Ведущий знак и пробелы не генерируются (как это делает функция STR$ ( )). Кроме того, ведущие нули отбрасываются. Таким образом аргумент 3 будет иметь результат "11"— другими словами, строку из двух символов, которые оба имеют в ASCII значение 31 равное 49 в десятичном виде. С учетом сказанного, максимальная длина строки результата функции BIN$ имеет длину 16 байт.

Если аргумент находится между нулем и 65535, то преобразование производится как для числа без знака

BIN$ (0) → "0"
BIN$ (65535) → "1111111111111111"

Однако, отрицательные аргументы от —1 до —32768 рассматриваются как числа в дополнительном коде (именно так система хранит целые числа).

BIN$ (—1) → "1111111111111111"
BIN$ (—32768) → "1000000000000000"

Следует заметить, что BIN$ (32768) → "1000000000000000" Для отрицательных аргументов BIN$ (отрицательный аргумент) = BIN$ (65536 + отрицательный аргумент). Очевидно, если необходимо преобразовать 16—битные целые числа, записанны только в дополнительном коде, нужно проверить аргумент и отклонить его, если он больше 32767. Функция VAL ( ) предусматривает обратное преобразование:

in = VAL ("&B" + BIN$ (in))

Использование: во-первых, эта функция может проверять являются ли числа нечетными:

DEF FN OD (in) = RIGHT$ (BIN$ (in), 1) = "1"

во-вторых, BIN$ ( ) используется для форматирования выводимого текста, при перемещении его в область буфера, FIELD, а также при выводе на экран или на печать. Наконец, эта функция помогает выполнять двоичную арифметику и вывод результатов, делая такие преобразования "безболезненными". Математические средства MSX—Бейсика, такие, как эта функция позволяют делать вычисления с разными без малейшего труда, особенно для прямого режма:

```
? BIN$ ( ( &B1011001000111100 — &H0A2C) OR
555)
1010101000111011
OK
```

Одной из проблем при этом является обязательный вывод ведущих нулей для того, чтобы гарантировать, что результат всегда будет иметь определенное число разрядов. Например, если выводимое поле должно быть всегда длиной в 16 бит, нужно сделать следующее:

DEF FN BN$ (in) = RIGHT$ (STRING$ (16, "0") + BIN$ (in), 16)

Пример:

```
10 PRINT "добавить два двоичных числа (максимально 16 бит) :"
20 INPUT "первое число, второе число"; N1$, N2$
40 ANS = VAL ("&B" + N1$) + VAL ("&B" + N2$)
50 BN$ = RIGHT$ (STRINGS (16, "0") + BIN$ (ANS), 16)
(или 50 BN$ = BIN$ (ANS),если длина BN$ безразлична)
60 PRINT "ответом является"; BN$; "или"; ANS; "десятичное".
```

См. также: OCT$ ( ), HEX$ ( ), STR$ ( ), VAL( ).

Цель: загрузка двоичного модуля, содержащего машинный код или данные, в память.

Формат: BLOAD *устройство — имя файла* [, *режим запуска*] [, *адрес смещения*]

Устройство — имя файла :: = любое строковое выражение, задающее устройство и файла (задание одного из 2-х обязательно).
Режим запуска :: = Буква R — для автоматического запуска после загрузки.
Адрес смещения :: = любое целое выражение

Действие: Эта команда, которая может находиться в программе, читает с заданного устройства двоичный файл, имя которого задано в команде. Устройство/ имя файла задается как обычно, в виде имя устройства: имя файла, где именем устройства является или CAS: (кассета), или внешее определенное имя во вставляемой кассете адаптера устройства (Например, A : B: могут обозначать дисковые устройства). Максимальная длина любого внешнего имени устройства в системе — 15 символов плюс двоеточие (:). Если имя устройства отсутствует, необходимо указать имя файла. Устройством по умолчанию будет CAS:. Допустимое имя файла зависит от устройства; для кассеты (которая является единственным устройством файлового ввода/ вывода подключаемым без адаптера) имя может иметь до шести алфавитно-цифровых символов, причем первый символ должен быть буквой. Дисковые файлы могут иметь 8-и символьные имена, после которых идет точка, затем трех-символьное расширение — максимально допустимое. В общем случае имя файла может быть да II символов без расширения, см. BSAVE. Если имя файла отсутствует, то используется первый найденный файл, который имеет двоичный тип.

Файл содержит начальный и конечный адреса и адрес запуска (созданные BSAVE), опередеkенные для размещения файла в памяти и для начала выполнения — режим R (RUN) будет выполнять передачу на адрес начала выполнения после успешной загрузки. Если есть адрес смещения, то но добавляется к этим значениям для помещения модуля в любое место ОЗУ.

Ввод с ленты и других устройств идет прямо в определенную область ОЗУ. Если не указать устройство/имя файла, то это вызовет ошибку "отсутствие операнда" "missing operand". если любой числовой параметр больше 65535, то произойдет переполнение, "overflow"
Можно остановить процесс загрузки, нажав клавиши Control—STOP, что вызовет ошибку на устройстве ввода/вывода, "Device I/O error".
При загрузке с ленты каждое просматриваемое имя файла идентифицируется сообщением SKIP: имя фойла, а загружаемый файл идентифицируется сообщением Found: имя файла. Эти сообщения подавляются в режиме выполнения, который идентифицируется значением систем ной переменной CURLIN (F41C) < > 65535.

Использование: так как эта команда может выполняться без использования стека переменных, она хорошо подходит для загрузки данных и оверлейных модулей прямо из выполняемой программы. Она позволяет также загружать непосредственно в оперативную память подпрограммы, написанные в машинных кодах для обращения к ним с помощью вызовов оператора USR. (Вы можете по желанию защитить эти подпрограммы. Для этого надо использовать оператор CLEAR. nnnn. Если при записи подпрограммы для начального и конечного адресов выл задан ноль, а адрес начала выполнения был задан смещением относительно начала, то при использовании оператора BLOAD адрес, заданный в "смещении", будет использоваться в качестве абсолютного адреса при загрузке модуля (в предположении, что он перемещаем). В любом случае адрес выполнения можно получить так:

DEF USR1 = PEEK (&HFCBF) + 256* (&HFCC0)

Этот адрес потом можно использовать только из Бейсик программы, а для инициализации при загрузке можно использовать опцию R.
С помощью PEEK (&HFCBE) = "R" можно установить была ли инициализация. И, наконец, с помощью BLOAD можно сохранить или восстановить состояние самого интерпретатоа Бейсик, т.е. содержание переменных ОЗУ и состояние ловушек. Нет нужды напоминать, что необходимо тщательно контролировать все выполняемые действия, однако этот оператор предоставляет некоторые возможности для зашиты от копирования.

Использование программных перекрытий в Бейсике означает, что модуль должен быть определенной длины, иметь номера строк в указанном диапазоне и должен быть создан, начиная с данного адреса ОЗУ. Размер такого модуля должен быть меньше размера модуля в исходной программе. Только в этом случае загрузки подгрограмм в машинных кодох можно выполнять с помощью оператора BLOAD, а затем обращаться к этим подпрограммам с помощью операторов GOSUB и GOTO. При этом последний оператор в модуле перекрытия должен оканчиваться тремя байтами, содержащими нули.

Если до использования BLOAD нужно определить длину модуля, то надо открыть файл и прочитать первые 4 байта. После этого можно будет вычислить длину, но для чтения файла придется выполнить обращения к ПЗУ. Хранение параметров файла подробныее описано в BSAVE.

Пример:

```
BLOAD "CAS: data", &HF400
Skip: test
Found: data
OK
BLOAD "CAS:", R
Found: prog
```

(Первый доступный файл был "найден", загружен и выполнен)

Смотри также: CLEAR, BSAVE, INPUT$ ( ), резделл ввод/вывод файлов.

Цель: Запись из оперативной памяти в файл машинных кодов или данных в двоичном виде.

Формат: BSAVE *устройство — имя файла, начадр, конадр* [, *западр*]

устройство — имя файла :: = строковое выражение, задающее имя устройства и (необязательно) имя файла
начадр :: = беззнаковое целое арифметическое выражение
конадр :: = беззнаковое целое арифметическое выражение
западр :: = беззнаковое целое арифметическое выражение

Действие: Содержимое ОЗУ, начиная с *начадр* и кончая *конадр*, записывается в файл с указанным именем на заданное устройство. В этот же файл записывается *западр* (если он задан). *Западр* — это адрес, с которого будет запускаться на выполнение программа после загрузки из файла с помощью команды BLOAD. Устройство — имя файла обязательно содержит имя устройства и (необязательно) имя файла. Подробности см. в описании команды BLOAD.

Если какой-либо из параметров пропущен, то будет выдано сообщение об ошибке "Missing operand," "пропущен операнд" или "Syntax error", "синтаксическая ошибка". Если в списке параметров присутствует западр, то он копируется также в системную переменную SAVENT (FCBF); в противном случае туда копируется начадр. Если значение любого из параметров превысит 65535, то будет выдано сообщение об ошибке "Overflow", "переполненной". Выполнение команды BSAVE можно прервать с помощью Control-STOP. В этом случае будет выдано сообщение "Device I/O error", "ошибка на устройстве ввода/вывода". Для кассетного

магнитофона прерывание этой операции произойдет с шестисекундной задержкой, поэтому будьте осторожны, чтобы случайно не стереть файл.

Последний адрес программы (*конадр*) запоминается в слове SAVEND (F87D), но начальный адрес (*начадр*) не сохраняется. Все данные в файл записываются блоком. Первые два байта — адрес начала, вторые два байта — адрес конца, третьи два байта — адрес запуска, после этого записывается содержимое ОЗУ. Запомните, что в двухбайтовом адресе слева стоит младший байт.

Команду BSAVE можно выполнять и в программе. Имя файла может содержать до 11 знаков (среди них не должно быть пробелов); имя устройства, если оно не является зарезервированным, может содержать до 15 знаков.

Использование: Эта команда выполняет действия, обратные команде BLOAD, в описании которой можно выяснить подробности. Чаще всего эта команда используется для сохранения областей оперативной памяти путем записи их в двоичном виде в файл на устройство ввода/вывода. За исключением накоторых переменнымх ОЗУ всё, в том числе и состояние Бейсик-интерпретатора или его частей, можно сохранить в файле. Адреса могут быть или абсолютными адресами, или иметь значение ноль (с указанием адреса запуска относительно начала программы). В последнем случае программа, содержащая внутри себя команду BLOAD, должна решить, куда поместить загружаемые данные (если они перемещаемы).

Пример:

```
BSAVE "CAS:", ST, ND
Ok
BSAVE "CAS:prog", &HF000, &HF0FF, &HF000
Ok
```

См. также: BLOAD, раздел "ввод/вывод файлов"

Цель: Позволяет использовать внешние, расширяющие возможности Бейсика, операторы, под программы которых, хранятся в ПЗУ.

Формат: CALL *имяпрог* [(*список параметров*)]

имяпрог :: = имя подпрограммы, состоящее из 15 (или менее) знаков

список параметров :: = необязательный список, содержащий одно или несколько выражений, отделенных друг от друга запятыми. Список, если он есть, обязательно заключается в круглые скобки.

ЗАМЕЧАНИЕ: в место "CALL" можно использовать символ подчеркивания "_".

Действие: Этот оператор позволяет расширить возможности системы MSX-Бейсик, находящейся в ПЗУ, путем добавления новых операторов, под программы которых, находятся в адаптере внешнего устройства или в кассете ПЗУ. При выполнении этого оператора Бейсик пытается найти такой "внешний оператор." Если в ПЗУ будет найдена подпрограмма с таким именем, то выполняется синтаксический разбор оператора. Это означает, что после имени подпрограммы может идти любой текст. Интерпретация списка параметров, идущего после имени программы, целиком возложена на подпрограмму, хранящуюся во внешнем ПЗУ. Вам, вероятно, следует избегать использования текста, не заключенного в кавычки и содержащего зарезервированные слова, так как они будут усечены до своих лексем; это же будет и с числовыми константами. Однако, если вы можете точно предсказать результат, то можно писать что угодно. Если же внешний оператор с указанным именем не будет найден, то будет выдано сообщение "Syntax error", "синтаксическая ошибка".

Использование: Любой разработчик программного обеспечения может поместить часть своей программы (особенно, если она в машинных кодах) в ПЗУ и обращаться к ней с помощью оператора CALL. Любое внешнее устройство также может содержать ПЗУ, в котором записана программа, управляющая этим устройством. Например, контроллер кассетного видео магнитофона может содержать оператор:

CALL VCR (ON) или CALL VCR (OFF)

Интерпретация всего, что идет после оператора CALL, полностью определяется подпрограммой, записанной в ПЗУ.

Пример:

(следующие два оператора эквивалентны)

```
CALL   РАДИОПЕРЕДАТЧИК (част, 128, "Привет", CW)
_РАДИОПЕРЕДАТЧИК (част, 128, "Привет", CW)
```

См. также: Приложение по стандарту на интерфейс расширителя магистрали.

Цель: Выполняет преобразование числа в число с двойной точностью

Формат: *x* = CDBL (*арвыр*)

арвыр :: = произвольное арифметическое выражение

Действие: Эта функция включена для обеспечения совместимости с другими версиями МБейсик, обладающими меньшими вычислительными возможностями. Вычисляется значение аргумента, и, если оно находится в допустимых пределах, то оно преобразуется в формат с двойной точностью: четырнадцать десятичных цифр. Однако, эта функция неявно выполняется каждый раз, когда присходит присвоение значения переменной двойной точности. Более того, так как Бейсик получает все результаты в этом формате, то можно не беспокоиться, что произойдет ошибка преобразования. Короче говоря, эта функция не выполняет ничего полезного. Последняя (пятнадцатая, считая слева направо) цифра округляется по правилу 5/4 — это значит, что, если отброшенная цифра была 5 или более, то к самой младшей из оставшихся цифре прибавляется единица. Если параметр отсутствует то будет сообщение "Syntax error", "синтаксическая ошибка". Если полученное число слишком велико (т.е. при округлении мантиссы экспонента превысит допустимое значение), то будет сообщение "Overflow", "переполнение".

Для хранения аргумента/результата используется ячейка DAC (F7F6)] Заметьте, что эта функция не преобразует тип переменной наподобие функциям MKD$ ( ) и CVD ( ), имеющимся только в дисковой версии языка Бейсик на кассете ПЗУ.

Использование: Основное назначение этой функции — обеспечить совместимость с другими версиями языка Бейсик. Например, некоторые версии МБейсика имеют малоэффективные программы преобразования для чисел с двойной точностью, хранящихся в памяти не в десятичном виде. Поэтому следующая конструкция

```
10 DEF INT A: A1 = 2: A2 = 3
20 X = CDBL (A1 / A2)
```

может оказаться необходимой, чтобы гарантировать от небольших ошибок, которые могли бы сказаться на окончательном результате. В MSX-Бейсике такой проблемы не существует, но использование этой функции позволит переносить существующие программы без ошибок.

Пример:

```
? CDBL (2 / 3)
.666666666666667
Ok
```

См. также: CSNG ( ), CINT ( ), форматы представления чисел в главе 2.

Цель: Преобразование целого числа в однобайтовый строковый символ

Формат: *x$*= CHR$ (*арвыр*)

арвыр :: = арифметическое выражение, причем 0 = <*арвыр*< = 255

Действие: Это функция преобразования типа. Вычисленное значение арифметического выражения должно быть между 0 и 255, иначе будет выдано сообщение об ошибке "Illegal function call", " недопустимый вызов функции". Над вычисленным значением неявно выполняется функция INT ( ). После этого создается однобайтовая строка, содержащая символ, соответствующий полученному коду. Другими словами, ниже приведенное равенство есть тождество:

CHR$ (65) = "A" (эти два элемента взаимозаменяемы)

Это означает, что строка, созданная функцией CHR$ (65), состоит из одного байта, причем биты этого байта образуют двоичный код "01000001", соответствующий десятичному числу 65. Отметим, что CHR$ (B01000001), CHR$ (&O81) и CHR$ (&H41) также взаимозаменяемы. Более того, чтобы для удобства чтения вывести, например, код Control—D, вполне допустимой будет следующая конструкция:

PRINT CHR$ (ASC ("D") — ASC ("@")) 'Ctrl — D = 04

Функцию CHR$ ( ) можно использовать везде, где допустимо использование однобайтовой строковой константы или строковой переменной. Обратной функцией является функция ASC ( ):

x = ASC (CHR$ (*x*)) и *x$* = CHR$ (ASC (*x$*))

Если значение x находится в допустимых пределах, то вышеприведенные равенства есть тождества.

Использование: CHR$ ( ) используется совместно с обратной функцией ASC ( ) и весьма полезна для хранения и восстановления небольших величин, т.к. допускает компактное их представление в виде однобайтовых строк. Для формирования многобайтовых строк используется конкатенация:

```
10 ANS$ = " " : FOR X = 1 TO 255
20 PRINT "Дай ответ"; X; " (0—255)";
30 INPUT ANS : ANS$ = ANS$ + CHR$ (ANS) : NEXT
или
10 ANS$ = CHR$ (X1) + CHR$ (X2) + CHR$ (X3)
...
```

Эта функция также очень полезна для решения многих небольших задач, возникающих все время при написании программ на Бейсике, особенно, если требуется инициализация. Например, SPRITE$ = псевдопеременная требует наличия символьной строки из 8 или 32 байтов, содержащей побитное описание объекта. Эта функция, часто используетая для инициализации:

SPRITE$ (2) = CHR$ (&HFF) + STRING$ (6, &H81) + CHR$ (&HFF)

инициализирует спрайт 2 в виде квадрата. Описание этого метода см. в описании псевдопеременной SPRITE$. Следует отметить, что сгенерированные таким образом коды не должны совпадать с кодами функциональных клавиш редактирования такими, как например, Control —E, вводимыми с клавиатуры в прямом режи-

ме; эти функции нельзя выполнить в режиме вывода наподобие PRINT CHR$(5).

И, наконец, иногда надо целое число представить в строковом виде для совместимости с некоторыми другими функциями. Запомните, что &HOC не вырабатывает код перевода страницы, а создает лишь число 12. Перевод страницы (код ASC II) можно получить только с помощью функции CHR$ ( ).

Пример:

```
PRINT #1, NAME$ + CHR$ (13) + ADDR$ + CHR$ (13) + CITY$
(CHR$ (13) это код "Возврат каретки", удобный в качестве разделителя полей)

ATT$ = "Внимание! Опасность!" + CHR$ (7): PRINT ATT$
(после вывода сообщения будет звуковой сигнал)

IF INPUT$ (1) = CHR$ (27) THEN GOSUB 2000
' нажата клавиша ESC
```

См. также: ASC ( ) , STRING$ ( ) , STRING$, кодов символов в приложении.

Цель: Преобразование арифметического выражения в 16-и битное число в дополнительном коде

Формат: *x* = CINT (*арвыр*)

арвыр :: = арифметическое выражение
-32768 = < арвыр < = 32767

Действие: Вычисляется значение аргумента и дробная часть результата отбрасывается. Если значение выражения выходит за границы диапазона представления целых чисел (от -32768 до 32767), то будет сообщение об ошибке "Overflow", "переполнение". В противном случае функция возвращает в качестве результата целое число. Эта функция неявно выполняется каждый раз, когда значение присваивается переменной целого типа или когда требуется аргумент целого типа. Так как в математическом пакете, входящем в состав системы MSX-Бейсик, редко возникают математические ошибки, то этой функцией пользуются нечасто.

Использование: Эта функция удобна для проверки типов чисел при вычислениях, особенно, когда у вас нет под рукой целой переменной. При этом для обработки ошибок вы сможете использовать оператор ON ERROR GOTO. По умолчанию диапазон целых чисел со знаком, представленных в дополнительном коде, соответствует вышеприведенному. Однако целые беззнаковые константы часто используются для задания адресов. Если вы хотите хранить целое без знака, т.е. в диапазоне от 0 до 65535, то нужно написать следующее:

```
x = CINT (va + (va>32767)*65536)' va-значение
```

Это позволит получить нужное число. Однако, в этом случае не будет зафиксирована ошибка, если число превысит 65535. Если переменная

или функция, получающие преобразованное число, не имеют целый тип, то значение заведомо неверное будет запомнено в десятичной форме. НО, если вы используете целочисленную переменную, то функцию CINT ( ) можно опустить. Таким образом, эта функция имеет ограниченное использование.

Пример:

```
100 ON ERROR GOTO 5000
110 PRINT CINT (X); "это первое значение"
.
.
.
5000 BEEP: INPUT "Слишком велико, введите заново"; X: RESUME 110
```

См. также: CDBL ( ), CSNG ( ), INT ( ), раздел "переменные" в главе 2.

Цель: Рисование окружностей и дуг любого размера и цвета, используется в режимах SCREEN 2 и SCREEN 3.

Формат: CIRCLE [@][STEP] (*x*, *y*), *радиус* [, *цвет* [, *нач. угол* [, *кон. угол* [, *овал*]]]]

x :: = координата по горизонтали; целое со знаком
y :: = координата по вертикали; целое со знаком
радиус :: = расстояние от точки (x, y); целое без знака -32768 < = целое со знаком < = 32767
цвет :: = код цвета: 0 =< *цвет* < = 15 по умолчанию совпадает с текущим цветом изображения
нач. угол :: = начало дуги в радианной мере; по умолчанию равно нулю
кон. угол :: = конец дуги в радианной мере; по умолчанию равно 2* π -2* π = < углы < =+2* π , где 2* π = 6.28318 (одинарная точность)
овал :: = отношение осей; число одинарной точности без знака, обычно близкое к 1; практически используются значения от 1/260 до 260

(любой необязательный параметр может быть опущен, однако соответствующая запятая в тексте должна указать на его отсутствие)

Действие: Рисует окружность, сектор круга или дугу окружности (точнее ту часть изображения, которая видна на экране). Центр круга (X, Y) может быть расположен в любой точке координатной плоскости. При указании параметра STEP координаты центра вычисляются путем сложения указанных в опраторе координат с координатами графического курсора (относительное задание координат) . Начало окружности

лежит слева от центра; рисование выполняется в направлении против часовой стрелки). Если радиус круга задать отрицательным, то этот параметр воспринимается как большое целое, что большой ценности не имеет). На экране изображаются только те точки, координаты которых лежат в пределах (0—255, 0—191), отсчитывая от верхнего левого угла экрана.

При указании параметров *еач. угол* и *кон. угол* рисуется дуга окружности. Если любой из этих параметров отрицательный, то используется его абсолютное значение; при этом соответствующий конец дуги соединяется с центром линией радиуса. При указании параметра овал (который по умолчанию равен 1), рисуется *овал*, так при значении этого параметра, равном 2, рисуется овал с отношением горизонтальной и вертикальной осей равным 2:1 (вертикальная задается радиусом). Ориентация овала всегда либо горизонтальная либо вертикальная.

Ошибки в задании параметров влекут за собой сообщения "Overflow", "переполнение" или "Syntax error", "Синтаксическая ошибка". Если на экране долгое время ничего не происходит, это может означать, что вы задали слишком большой размер радиуса окружности, и она не видна на экране. Если вы воспользовались оператором CIRCLE в режимах SCREEN 2 и SCREEN 3 задаются одинаково; при этом в режиме SCREEN 3 каждому пикселю соответствует блок из 4x4 точек. Символ @ добавляется в текст оператора для совместимости с другими версиями Бейсика.

Использование: Это-единственный оператор MSX-Бейсика, позволяющий рисовать кривые без циклов; с его помощью окружности рисуются гораздо быстрее, чем с помощью тригонометрических функций. Оператор CIRCLE позволяет также строить

секторные диаграммы (при задании "отрицательных" значений параметров *нач. угол* или *кон. угол.*) Значение параметра "угол", равное — 0, не будет воспринято; следует пользоваться малым отрицательным числом (например, .001). Если круги на экране оказываются "недостаточно круглыми", пользуйтесь параметром "овал", выбирая его в диапазоне от 1 до 1.2. Для людей, непривычных к радианной мере, советуем пользоваться простым приемом:

```
DEF FN CD (радиан) = радиан*360/6.28318
```
или просто радиан*57.2958; можно также пользоваться этой константой прямо в выражениях.

Может также вызвать путанницу привычка отсчитывать углы принятым в математике образом: направо=0, вверх=90, налево=180, вниз =270. Значения углов не должны превышать 6.28318 рад. Но на экране координатная плоскость — другая: первый квадрант "отражается" вниз, поэтому начало координат — слева вверху, и все видимые точки имеют положительные координаты. *Нач. угол* и *кон. угол* могут быть заданы, отсутствовать или быть отрицательными.

Пример:

```
10 SCREEN 2: COLOR 9' секторная диаграмма
20 CIRCLE (100, 100), 75,, -1, -.001, 1.2
30 CIRCLE STEP (10, -5), 75,, -.001, -1, 1.2 ' Эта разорванный клин
40 GOTO 40 ' рисунок стоит на экране
```

См. также: LINE, DRAW, POINT, раздел "графическое программирование."

Цель: Удаляет переменные, отводит место для строк и конструкций, не входящих в программу на Бейсике

Формат: CLEAR [*размер — строки* [, *совб — место*]]

размер строки :: = число байтов, отводящееся для строк 0 = < *размер строки* < = свободная память

Своб — место :: = адрес верхней границы области памяти, используемой системой

Действие: Во-первых, при любой форме этого оператора удаляются все переменные: это означает, что удаляется все, кроме текста программы. Сюда входят массивы и переменные, определенные функции, строки символов и все содержимое аппаратного стека. Все файлы закрываются и с помощью вывода CR/LF (возврат каретки, перевод строки — прим. пер.) завершается вся печать, т.к. буфера очищаются. Все, что создается с помощью операторов DEF xxx (кроме DEFUSR) полностью пропадает. Все это разрушает единственная команда: CLEAR

Во-вторых, она используется для определения объема памяти, отводимой для хранения текста (по умолчанию он равен 200 байтам). Если задан размер — строки, то отводится указанное количество байтов.

Третье использование — при задании обоих аргументов. Обычно, при программировании на MSX-Бейсик об этом не стоит заботиться, Но тот, кто захочет использовать эту возможность, должен знать следующее: первый аргумент используется для определения области памяти для хранения строк (см. выше); второй — это новый адрес, заносящийся в слово HIMEM (адрес — FC4A), указывающий на первый байт памяти, который не будет обработан как программа или данные, и отводящий место для неформатированных данных, машинного кода и других видов информации, придуманных пользователем. При этом оператор CLEAR также удаляет и восстанавливает все FCB, занося в MAXFIL (F85F) 1 + наибольший номер файла. Заметим, что MAXFILES = также может сбросить это значение и заново создать область для хранения строк и т.д. также, как и оператор CLEAR.
См. карту памяти, приведенную в приложении.

Существует несколько ограничений; между STKTOP и концом текста программы Бейсик, указанным в слове VARTAB (F6C2) должно быть достаточно места' около 145 байтов (для массивов). Это ограничение от своб-место (т.е. HIMEM) до VARTAB вычисляется так:

свобместо — 267 * (макс — файл +1) — размер — строки — 130 > VARTAB

где макс-файл= PEEK (&HF859): MAXFIL.
Если MAXFILES равно 0, а размер — строки тоже нулевой, границей своб-место являются слова от BOTTOM (FC48) + (1B0) до (F380), иначе в нижней границе возникает ошибка "выход за пределы памяти", "Out of memory", а в верхней — "неправильная функция". "Illegal function call".

Если опустить только первый аргумент (с сохранением запятой), возникнет синтаксическая ошибка. Переполнение происходит, если никакой параметр не преобразуется к 16-битовому беззнаковому целому типу.

Примечание: CLEAR — это мобильный способ отведения памяти под информацию, не относящуюся к Бейсик-программе, и для строк. Фактически, область для хранения строк очень капризна:

если она очень мала, в непредвиденный момент можно выйти за границы памяти; если велика, можно выйти за пределы области для переменных. Возникает много "мусора". Вопреки общему мнению, отведение слишком большой области для строк только задерживает момент получения результата, а частая информация от FRE (." ") ничего не спасает. Требуемое время определяется только числом использующихся элементов строкового массива. Для устранения этой задержки используйте команды MID$ ( ) = SWAP и LSET/RSET для текущей работы со строками без рабочих областей.

Пример:

```
10 CLEAR 5000: DEF INT A–Z: DIM A$ (1500)
20 FOR X = 0 to 1500: A$ (X) = "2" + " ": NEXT
30 REM размещение данных в области для строк, мусора нет
40 TIME = 0: A = TIME: PRINT FRE (" "), (TIME – A) /50
```

В этом примере происходит ожидание в 3.5 мин. (TIME отсчитывает время). При изменении строки 10 на CLEAR 1600 получается почти то же время. Т.о., единственный способ уменьшить или исключить эту проблему — это использовать минимальное количество строк, и, особенно, строковых массивов (См. раздел "Операции над строками").

В CLEAR можно использовать переменные, только надо помнить, что они теряются при выполнении CLEAR. Можно выбрать подходящую системную переменную для временного хранения значений. И запомните, что если свобместо не задано, то это значение (т.е. HIMEM) не используется в CLEAR .

Многие операторы неявно используют CLEAR (например, RUN "имя", NEW, MERGE, успешно завершенный LOAD, MAXFILES и др. Этот оператор также неявно используется при изме-

нениях в тексте Бейсик-программы. Потери переменных можно восполнить копированием подходящих системных переменных в безопасные области и их последующим восстановлением (если новая программа не больше исходной) или переносом всего, кроме данных, вверх, а затем обратно после загрузки.

Пример:

```
CLEAR 1000, &HF000
```

(отводит 1000 байтов для строк и RAM с (F000) до (F380) для нужд пользователя).

См. также: MAXFILES =, карту памяти в приложении, раздел "Обработка строк".

Цель: Загрузка (или верификация) программы на языке Бейсик с ленты во внутренний формат.

Формат: CLOAD [?] [*имя-файла*]

имя-файла :: = строковое выражение, переменная или константа (в кавычках); имеют значение только первые 6 знаков, они могут быть любыми, кроме пустой строки.

Действие: Загружает данные с ленты. Если символ "?" отсутствует, происходит загрузка файла с кассеты, которая должна быть записана во внутреннем формате. На ленте стандартный заголовок, за которым следует 10 стартовых байтов, содержащих (D3) и указывающих на текст программы, и соответствующее имя файла, состоящее из 6 символов. Хранение имени файла, ошибки и поиск аналогичны оператору BLOAD (см. BLOAD). Пустое имя файла приводит к сообщению "Illegal function call", "Неправильный вызов функции", а останов загрузки с помощью Control-STOP приводит к выводу сообщения "Device I/O error", "ошибка устройства ввода-вывода".

При выдаче сообщения Found (найдено) выполняется команда очистки памяти NEW, и текст читается непосредственно в память из одного длинного блока данных.

Если присутствует символ "?", производятся те же действия, но команда NEW не выполняется; вместо этого производится побайтовое сравнение загружаемого текста и текста, занесенного в ОЗУ, куда должна была быть загружена программа. При первом же нессответствии, лента останавливается и выводится сообщение "Verify error", "ошибка верификации". Если сравнение прошло нормально, выполнение команды CLOAD заканчивается (с выдачей подсказки

OK в прямом режиме) при достижении конца программы на ленте.

Использование: Эта команда является основным средством загрузки уплотненной программы на Бейсике с ленты. Она также является быстрой. Не следует использовать команду LOAD "CAS:", т.к. для кассет она автоматически использует формат ASCII.

Пример:

```
CLOAD "test"
Found: test
Ok
CLOAD ? ".2File"
Skip : 3xx
Found: .2File
Verify error
Ok
```

См. также: CSAVE, BLOAD, раздел "программирование ввода/вывода файлов".

Цель: Завершает операции ввода-вывода и освобождает соответствующие буфера.

Формат: CLOSE [[#] *ном-буф*] [, [#] *ном-буф*] . . .

ном-буф :: = целое число, ссылка на буфер ввода-вывода,
1 = < *ном-буф* < = 15
(может быть равно 0, но это бессмысленно)

Действие: Если оператор CLOSE задан без аргументов, все открытые с помощью оператора OPEN файлы закрываются; может быть указан список номеров открытых файлов (с необязательным "#". Закрытие файла означает, что все данные, оставшиеся в буфере читаются/записываются, все операции ввода-вывода прекращаются и в каждый открытый файл печатающего устройства записываются коды CR/LF. Для файлов вывода вырабатывается признак конца файла.

Использование: Очень важно следующее: если файл является последовательным и не закрыт с помощью CLOSE, то признак конца файла не записывается; если же вы работали с файлом прямого доступа и не закрыли его с помощью CLOSE, то длина файла может оказаться неверной. Если файл не закрыт с помощью CLOSE, то может нарушиться структура каталога диска.

Оператор END выполняет неявный вызов оператора CLOSE для всех файлов так же, как и оператор CLEAR, включая автоматические действия при LOAD, MAXFILES =, NEW и действия, производящиеся при изменении текста программы.

Пример:

```
CLOSE: OPEN F$ FOR INPUT AS #2 ' "все закрывается, затем открывается 2"
CLOSE #2... ' закытие только 2, все остальное открыто".
```

См. также: OPEN, CLEAR, MAXFILES =, раздел "ввод-вывод файлов".

CLS

оператор

Цель: Очистка экрана в любом режиме

Формат: CLS

Действие: CLS устанавливает весь экран в цвет фона. Этот оператор применим к любому режиму и таким образом является самым быстрым способом получить чистый экран опеределенного цвета. Все уничтожается точно так же, как и при нажатии клавиши Shift-HOME/Clear; однако, состояние курсора и подсказки для функциональных клавиш остаются неизменными.

Использование: Оператор CLS не только быстрый, легкий и удобный; он также имеет несколько удобных свойств. В режиме SCREEN 1 (по умолчанию), если вы работали с таблицей цветов в VRAM для получения нескольких цветов на экране, то CLS не заносит в таблицу цвета изображения/фона, как это делает SCREEN. В режимах SCREEN 2 или 3 команда Color x, y не оказывает немедленного воздействия — для изменения цвета фона всего экрана на другой требуется оператор CLS.

Другое преимущество CLS — в "быстроте"; если на экране надо воспроизвести какие-либо расчеты, то очистка области, в которую могут быть выведены числа, быстрее всего может выполнена с помощью CLS:

```
10 OPEN "CAS:TEST" FOR INPUT AS #1
20 INPUT #1, A: CLS: PRINT "NEXT NUMBER READ:";A
30 IF NOT EOF (1) THEN 20 ELSE END ' "закрывает
   файл".
```

Пример:

```
IF OPTION = 0 THEN CLS : PRINT: PRINT : PRINT "Bye-Bye!": END
```

См. также: BASE, COLOR, SCREEN

оператор

COLOR

Цель: Установка новых цветов изображения/фона, принимаемых затем по умолчанию

Формат: COLOR [*цветном*] [, *цветном*] [, *цветном*]

цветном :: = цвет изображения, фона или границы, 0 = < *цветном* < = 15

(Замечание: некоторые параметры можно опустить при наличии необходимых запятых).

Действие: Установка цветов, принимаемых затем по умолчанию. Эффект зависит от текущего режима экрана.

Любой опущенный параметр, обозначенный запятой перед данным параметром, сохраняет старое значение. Ниже приводятся номера цветов:

0: прозрачный: позволяет совпасть цвету границы и оставшейся части экрана. (Если есть внешний видеовход, граница тоже должна иметь цвет 0).
1: черный
2: средне-зеленый
3: светло-серый
4: темно-голубой
5: светло-голубой
6: темно-красный
7; голубой
8: средне-красный
9: светло-красный
10: темно-желтый
11: светло-желтый
12: темно-зеленый
13: васильковый
14: серый
15: белый

(Замечание: ТВ-мониторы могут оказаться различными по цветовым характеристикам; разли-

чные сочетания цветов могут также дать необычный цвет на экране. Попробуйте сделать это).

Отсутствие параметров приводит к сообщению "Missing operand", "пропуск операнда", а слишком большое значение параметра — К сообщению "Illegal functio call", "неправильный вызов функции".

Использование: Конечно, изменение цвета текста. Но оператор COLOR также воздействует на все графические операторы: если цвет не задан, то берется последнее значение оператора COLOR. Таким образом можно написать подпрограммы для рисования нескольких объектов, а их цвет и расположение определять в главной программе перед выполнением оператора GOSUB. Заметьте: многие комбинации изображения/фона дают нечитаемый текст. Проверьте это! Если текст находится на краях поля, попробуйте изменить только цвет границы — это иногда помогает. Можно случайно установить один и тот же цвет для изображения и для фона, тогда экран будет пустым или ЭВМ не будет отвечать на команды.

Сброс цветов осуществляется нажатием клавиш Control-STOP и SHIFT HOME/CLS, а затем SHIFT F1/F6.

Пример:

```
COLOR, 7 : CLS ' установка голубого фона или близкого к нему, если
SCREEN 2
100 SCREEN 2: COLOR 8: GOSUB 1000 ' оранжевый квадрат
200 DRAW "BM100, 100": COLOR 10: GOSUB 1000 ' желтый квадрат
    снизу
300 GOTO 300
1000 LINE STEP (0, 0) – STEP (20, 20) ,,BF: RETURN ' рисует квадрат
```

См. также; CLS, раздел "Графическое программирование"

Цель: Продолжение выполнения программы с места выхода из нее

Формат: CONT

Действие: Возобновление выполнения программы с той точки, где она была прервана (с помощью STOP, END, Control-STOP). При продолжении выполнения программы с помощью CONT выполняется оператор GOTO старый номер строки. Если что-либо делает это продолжение бессмысленным, например, в случае оператора CLEAR, любая попытка выполнить команду CONT приводит к сообщению: "? Can't continue", "нельзя продолжить".

Применение: STOP/CONT — это основной способ отладки. Если вставить операторы STOP в нескольких местах программы, то при сообщении BREAK можно вывести значение переменных в прямом режиме, а затем набрать CONT. Программа не обнаружит разницу в выполнении: во время прерывания вы можете даже изменять значения различных переменных. Если при сообщении "? нельзя продолжить" вы хотите продолжить выполнение программы, используйте оператор GOTO.

Пример:

```
1050 GOSUB 9000: STOP: GOSUB 10000 ....
RUN
Break in 1050
Ok
? A$
shirley
Ok
CONT
.
.
.
```

См. также: STOP, END

COS

фукция

Цель: Вычисление косинуса угла в радианах

Формат: *x* = COS (*угол-рад*)

угол-рад :: = угол в радианах: любое арифметическое выражение

Действие: Эта функция вычисляет значение косинуса угла, определенного в качестве аргумента (в радианах). Заметьте, что для вычисления косинуса используется степенной ряд констант двойной точности; поэтому результат будет весьма точным. Например, погрешность в выражении:

```
PRINT COS (ATN (1)*4)
```

равна: 1E—14! При округлении до 13 значащих цифр можно построить тригонометрическую таблицу.

При отсутствии аргумента выводится сообщение "Syntax error", "синтаксическая ошибка", а нечисловой аргумент приводит к сообщению "Type mismatch", "несоответствие типов".

Применение: Везде, где требуется. Для перевода из градусов в радианы используется формула:

x*PI/180, где PI=ATN (1)*4 или 3,1415926535898

См. приложение "Производные функций"

Пример:

```
PRINT COS (0)
1
Ok
```

См. также: SIN ( ), TAN ( ), ATN ( ), приложение о тригонометрических функциях.

команда

CSAVE

Цель: Запись текста программы, написанной на языке Бейсик, на магнитную ленту во внутреннем формате.

Формат: CSAVE *имя-файла, скорость*

имя-файла :: = строковое выражение, используются первые 6 символов.

Скорость :: = арифметическое выражение, принимающее значение 1 или 2

Действие: Сохраняет текст Бейсик-программы, используя внутренний формат. Практически любую строку символов можно использовать в качестве имени-файла; но используются только первые 6 символов (имя, содержащее менее 6 символов, дополняется справа оробелами). Скорость передачи информации на ленту из соображений надежности равна 1200 бод (по умолчанию) и может быть увеличена до 2400 бод (это требует хорошей лентопротяжки). Скорость можно задать в операторе SCREEN или с помощью второго аргумента в CSAVE — 1 = 1200 bd., 2 = 2400 bd.

При отсутствии или неверном задании имени файла выдаются сообщения "Missing operand", "пропущенный операнд", "Type mismatch", "несоответствие типов" и "Illegal function call", "неправильный вызов функции".

Использование: Быстрый способ для сброса программы на ленту — см. CLOAD. Если использовать SAVE "CAS:", то для ленты это по умолчанию означает переход в режим кода ASCII. Внимание! действие клавиши Control-STOP подавляется на первые 6 с. после запуска команды: будьте осторожны при нажатии кнопки RECORD на лентопротяжке.

Пример:

```
CSAVE "1001", 2 ' сохранение на большой скорости
```

См. также: CLOAD, BSAVE, раздел "ввод-вывод файлов".

Цель: Приведение аргумента к типу одинарной точности с округлением

Формат: *x* = CSNG (*ар. выр.*)

ар. выр. :: = любое арифметическое выражение

Лейсивие: Эта функция вычисляет значение аргумента. Результат округляется до 6 значащих цифр с округлением по правилу 5/4: проверяется седьмая значащая цифра (счет идет слева направо), если она равна 5, то к шестой цифре прибавляется 1.

Если аргумент отсутствует, является строкой или имеет значение экспоненты больше или меньше допустимого, возникает синтаксическая ошибка, "Type mismatch", "несоответствие типов" или "Overflow", "переполнение". Эта функция используется редко.

Использование: Имеет важное применение, обеспечивая совместимость. Может использоваться для округления в прямом режиме или в операторе PRINT без введения лишней переменной; если требуется присвоить результат переменной типа одинарной точности, то эта функция не нужка. Например,

```
PRINT CSNG (a) ' выводит только 6 значащих цифр
с округлением.
```

Полезным является следующий прием:

```
x = CSNG (x)
```

это позволяет избавиться от цифр, введенных для уменьшения ошибок усечения и т.д.

Пример:

```
20 PRINT "Промежуточные результаты:"; CSNG (REG)
30 GOSUB 1000 ' выполняет другие вычисления над REG
40 REG = CSNG (REG) ' готовность выдать окончательный результат
```

См. также: CINT ( ), CDBL ( ), раздел главы 2 "Переменные".

Цель: Определение номера строки экрана, на которой находится курсор

Формат: *x* = CSRLIN

Действие: В режиме SCREEN 0 или 1 функция возвращает номер строки в виде целого числа. Этот номер указывает место нахождения курсора на экране. Верхняя строка имеет номер 0, и счетчик увеличивает свое значение при движении вниз по экрану до троки 22 или 23 (в зависимости от того, используется ли строка для индикации функциональных ключей).

Использование: Используется с функцией POS ( ) для определения точки экрана, в которую будет направлен последующий вывод. Функция LOCATE может использоваться для передвижения курсора. Для выполнения аналогичной функции в режиме SCREEN 2 или 3 используйте PEEK (&HFCB9).

Пример:

```
IF CSRLIN = 23 THEN PRINT X;: LOCATE 0, 1 ELSE PRINT X
```

(для предотвращения вертикального продвижения экрана)

См. также: POS ( ), LOCATE, раздел "Графическое программирование".

Цель: Хранение констант в тексте программы для использования по мере необходимости

Формат: DATA *элемент* [, *элемент*] . . .

элемент :: = числовая константа, строковая константа в кавычках или без кавычек, но не содержащая запятых, кавычек и двоеточий (, / " / :)

Действие: Оператор позволяет хранить "начальные" значения в теле программы. Оператор DATA аналогичен любому другому оператору; его можно редактировать, он может находиться в середине строки операторов и в строке операторов прямого режима (тогда он игнорируется). Любые служебные слова языка Бейсик или цифровые константы внутри DATA преобразуются в лексамный внутренний формат, но при считывании данных оператором READ они преобразуются к первичному формату (как при LIST) для предотвращения двусмысленностей.

Основной характеристикой оператора DATA является его полное игнорирование и пропуск во время выполнения программы — он используется только в операторе READ. Операторы DATA считаются частью общего "банка данных" программы. Поэтому 10 операторов DATA, каждый из которых содержит один элемент, разбросанные по всей программе, неразличимы с одним оператором DATA, содержащим все элементы данных из этих операторов. При чтении данные берутся из самого первого оператора DATA (в смысле номеров строк), и все последующие операторы DATA выбираются по очереди. При "исчерпании" всех операторов DATA данные не могут считываться, пока на выполнен оператор RESTORE. RESTORE позволяет выбрать любой нумерованный оператор DATA в качестве "следующего" для чтения.

В каждом наборе элементов данных оператора DATA запятая (,) используется в качестве разделителя; типы элементов данных могут быть различными.
Апостроф не считается признаком конца оператора и началом комментария: в операторе DATA такую роль играет двоеточие (:).

Допускается использование любой числовой константы (но не выражений или переменных): константы могут включать &H, &O, &B или быть в экспоненциальной форме; константы могут иметь указатель типа. Эти константы могут читаться в ячейки ранее заданных числовых констант, в отличие от других типов элементов данных. Строковые константы могут заключаться в кавычки и если важно, чтобы каждый символ элемента данных хранился в первоначальном виде (в коде ASCII), то он также должен заключаться в казычки. Строковые константы можно читать оператором READ только в строковые переменные. Наконец, в том случае, когда в строковой константе отсутствуют запятые, кавычки можно опускать. В этом случае система пытается минимизировать объем рабочей памяти с помощью образования лексем. Любая синтаксическая ошибка, вызванная несоответсивием данных в операторе DATA, выдает номер ошибочной строки оператора DATA.

Использование: Оператор DATA — это самый краткий способ инициализации констант в программе, а также единственный практичный способ инициализации массивов. Он очень удобен для документации.

Другое его применение — в инициализации спрайтов; это помогает представить, что вы рисуете.

Пример:

```
400 DATA 11111111 : ' начало квадрата
401 DATA 10000001
402 DATA 10000001
403 DATA 10000001
404 DATA 10000001
405 DATA 10000001
406 DATA 10000001
407 DATA 11111111 : ' конец квадрата — режим 8x8 точек
410 RESTORE 400: SP$ = " "
420 FOR SB = 1 TO 8: READ SB$
430 SP$ = SP$ + CHR$ (VAL("&B"+SB$) : NEXT
440 SPRITE$ (17) = SP$ ' присваивает спрайт #17 квадрату
```

Обратите внимание на использование оператора RESTORE для выбора "именованного" (по номеру строки) блока операторов DATA — это хароший пример логичного размещения операторов и их компонентов.

Пример:

```
10 DATA 1, 2, 3: BEEP
20 FOR X = 1 TO 10: DATA &H4: READ A(X): NEXT
30 DATA &B101, 6, 7!, &O10, 9, 10e + 0
```

См. также: READ, RESTORE.

Цель: Определение функции задаваемой, пользователем

Формат: DEF FN *имя-перем* [(*арл* [, *арл*] . . .)] = *выражение*

имя-перем. :: = любое допустимое имя переменной, для которого выделяется память;
арг. :: = любое допустимое имя переменной; память не выделяется
выражение :: = любое выражение, имеющее тот же тип, что и арг.

Действие: Создает определение функции, заданной пользователем. При выполнении этого оператора создается переменная *имя-перем.*, которая является именем функции (помеченным установкой старшего бита первого символа имени). Вычисления в момент определения не производятся; оператор DEF FN может находиться в любом месте строки. В памяти под эту переменную (имеющую тип, определенный по умолчанию или в имя-перем) отводятся два байта для адреса символа "=" в тексте этого определения. К этой переменной обычный доступ запрещен, однако, с этим же именем можно создавать и использовать другую переменную, причем эти две переменные будут различимы.

Если определена другая функция с тем же именем, старое определение теряет силу. Функция пользвателя, определенная именем *имя-перем.*, может иметь любой тип; это означает, что значение, которое должно вычисляться функцией, будет преобразовано к типу переменной *имя-перем.* Функция может не иметь аргументов вообще или иметь один или несколько аргументов. Они определяются как список фиктивных имен переменных любых типов, разделенных запятыми и заключенных в круглые

скобки после *имя-перем*. Каждый *арг* является формальным параметром: это означает, что он на самом деле "не существует" и не будет взаимодействовать с переменными с тем же именем в программе. Он создается временно для самого выполнения функции пользователя и затем при возврате из функции удаляется. Каждый *арг* должен иметь тип, который может быть присвоен любому возможному значению, принимаемому аргументом.

После символа равенства (=) стоит любое выражение. Оно вычисляется при вызове функции пользователя, и результат после преобразования к типу, определенному переменной *имя-перем*. возвращается в точку вызова функции. Если это выражение содержит имена переменных, описанных в качестве параметров, их значения вычисляются из списка фактических параметров основной программы — соответствующее входное выражение было вычислено и присвоено временной переменной имя-перем.; это значение используется в вычислении функции. Если используется другое имя переменной, берется ее значение из основной программы. В выражении, допустим, любой вызов функции, включая PEEK ( ) и нерекурсивные вызовы других функций, заданных пользователем.

Выражение не вычисляется до тех пор, пока не произошел вызов функции.

В процессе определения функции не ведется никакой проверки: она начинается при фактическом обращении к функции. Если в строке, где вызывается функция пользователя, содержится ошибка, то проверка определения функции может эту ошибку выявить.

Использование: Это единственный способ избавить вашу программу от вопроса о "нумерации строк" — изме-

нить порядок выполнения программы без знания номеров строк по-другому нельзя. Однако, DEF FN в отличие от DATA, — выполняемый оператор, функция должна быть определена *до* ее запуска. Поскольку текст определения функции хранится в памяти как часть программы, DEF FN — один из нескольких операторов, к которым нельзя обращаться в прямом режиме.

Отметим, что с одним именем можно определить 4 функции (3, если вы работаете с возвращаемыми значениями, являющимися 16-битовыми числами со знаком), различающиеся по типам переменных. Тогда можно вызвать функцию по имени без упоминания о ее типе и таким образом вызвать любую из них, основываясь на установке типа имени по умолчанию:

```
10 DEF FN Q% = 1: DEF FN Q! = 2: DEF FN Q = 3
20 DEF INT Q: PRINT FN Q: DEF SNG Q: PRINT FN Q
30 DEF DBL Q: PRINT FN Q: PRINT FN Q%; FN Q!; FN Q
```

Это приводит к удивительным последствиям: например, перехват по ошибке может обнаружить несоответствие типов в вызове функции, изменить функцию — даже строковую и пересчитать ее значение. Или, программа может сама себя модифицировать с помощью динамического изменения вызова функции для отражения этапов работы. Возможно, существуют еще большие применения этой особенности.

Пример:

```
10 DEF FNQU$ ( QU, ST ) = MID$ ( QL$ (QU), ST )
20 PRINT "Next Question"'; FNQU$ ( CO,LN (CO) )
```

См. также: FN, DEFINT/SNG/DBL/STR.

DEF INT / SNG / DBL / STR

оператор

Цель: Объявляют типы по умолчанию для групп имен переменных

Формат: DEF *тип-пер. диап.* [, *диап.*] . . .

тип-пер. :: = INT, SNG, DBL, или STR
диап. :: = буква алфавита или две буквы, разделенные дефисом (—), представляющие диапазон букв алфавита (по порядку).

Действие: Это оператор объявления, определяющий типы переменных по умолчанию. Весь набор имен переменных, начинающихся с букв алфавита, которые заданы аргументом *диап.*, будет иметь тип *тип-пер.* Исключение составляет случай, когда переменные используются с идентификатором типа (#, $, % и !).

Использование: Этот оператор следует применять в каждой программе: при запуске программы с оператором DEFINT A—Z гарантируется минимум трехкратное ускорение ее выполнения. Запомните, что это выполняемый оператор, он работает только с именами переменных, не имеющих явного указания типа.

Пример:

```
DEF INT A—Z: DEF SNG C, E, G—J: DEF DBL D: DEF STR Q
```

См. также: DIM, CLEAR, раздел главы 2 о хранении переменных, Приложение "карта памяти".

DEF USR

Цель: Определяет адрес вызываемой пользователем программы в машинном коде.

Формат: DEF USR [*цифра*] = *цел-выр.*

цифра :: = одноразрядное число; если оно отсутствует — подразумевается 0
цел-выр :: = выражение обычно целого типа в диапазоне 32768 = < *цел-выр.* < = 65535

Действие: Оператор определяет вход в определенную пользователем функцию-подпрограмму в машинном коде. Вычисляется выражение; результатом является адрес начала программы. Выражение считается 16-битовым целым без знака; если оно отрицательно, то считается, что адрес больше, чем 7FFF: -32768 ставится в соответствие адрес 32768, а -1 — адрес 65535. Если выражение положительно и больше 32767, оно преобразуется в соответствующее целое (см. выше), так, что адрес может задаваться более простым способом, например, любое выражение, равное 49000 фактически представляет в памяти адрес 49000. Если задана константа со значением большим, чем 32767, она хранится как специальное значение одинарной точности (выводится на листинг с символом "восклицательный знак" в конце числа) для того, чтобы правильно ее интерпретировать.

Соответствие между вызовом USR ( ) и адресом задается с помощью этого оператора путем занесения вычисленного адреса в слово памяти, находящееся в области USRTAB (F39A), которая содержит по 2 байта на каждый из 10 возможных вызовов USR ( ), определенных одновременно. Номер функции USR ( ) задает *цифра* в команде DEF USR; если цифра отсутствует (для совместимости со старыми версиями языка МБейсик), то по умолчанию производится вызов функции с номером 0.

DEF USR

оператор

Использование: Это выполняемый оператор, который можно имитировать путем записи в область USRTAB, а также с помощью программ в машинном коде. Он может связывать программы, написанные в машинном коде, с Бейсик-программой. Для выполнения программ в машинном коде необходимо сделать следующее:

```
DEF USR = &HXXXX: a = USR (a)
```

Детали методов вызова и передачи параметров см. в описании USR.

Можно записывать программы в стандартные области ОЗУ и указывать их местоположение для оператора DEF USR. Например, BLOAD сохраняет стартовый адрес загруженной программы в SAVENT (FCBF), поэтому следующие операторы являются логически верными:

```
100 BLOAD "CAS: sort"
110 AD = PEEK ( &HFCBF ) + 256 * PEEK ( &HFCC0 )
120 DEF USR 1 = AD
```

Функция CLEAR сохраняет значение USRTAB вместе со всеми определенями.

Пример:

```
10 DEFUSR=60000!: DEF USR 3 = ADDRESS + 2
```

См. также: USR ( )

команда

DELETE

Цель: Удаление строк текста программы из памяти.

Формат: DELETE *диап-строк*

диап-строк ::= номер строки, или диапазон номеров строк, или дефис, за которым следует номер строки.

Замечание: Вместо номера строки может стоять точка (.); она обозначает "текущую" строку.

Действие: Эта команда может удалить часть текста или всю программу, поэтому ее надо использовать осторожно. Любой номер строки (или номера группы строк, представленные в виде; номер начальной строки — номер последней строки) может стоять 6 DELETE. Номер начальной строки можно опустить; в этом случае берется первая по порядку строка. Для ссылки на номер текущей строки можно использовать точку (см. описание экранного редактора) . Если номер строки в программе отсутствует, выдается сообщение "Illegal function call", "неправильный вызов функции" — это единственное средство защиты в команде DELETE.

Использование: При использовании команды DELETE выполняется оператор CLEAR. Более того, из программы вы вернетесь в прямой режим с подсказкой "Ok". Этого можно избежать методом "псевдо-клавиатуры", в котором команда заносится в буфер служебных слов и таблицы переменных восстанавливаются с помощью указателей.

При удалении строк с помощью DELETE удаляются также все указатели, сформированные в тексте Бейсик-программы для этих строк.

DELETE

команда

Пример:

```
DELETE 100–200
DELETE .
DELETE –5000
DELETE 50–.
DELETE 135
```

См. также: CLEAR, MERGE

оператор

DIM

Цель: Объявление переменных, особенно массивов с инициализацией нулями.

Формат: DIM *имя-перем.* [, *имя-перем*] . . .

имя-перем. :: = любое имя переменной, имеющей или не имеющей тип, за которой в круглых скобках может следовать список максимальных значений индексов.

Действие: Этот оператор создает переменные, отводя для них место в памяти и присваивая им начальное значение 0. Он обычно используется для объявлять простые переменные. Форматы памяти для переменных описаны в разделе главы 2 и в приложении "Карта патяти".

Каждый созданный массив имеет количество элементов, равное произведению максимальных индексов ("границ"), к каждому из которых прибавляется 1, т.к. минимальное значение индекса равно 0. При создании простой переменной, все переменные массива продвигаются в памяти для освобождения места.

Если после DIM ничего не стоит, возникает синтаксическая ошибка; попытка применить оператор DIM к уже существующему массиву приводит к ошибке "Redimensioned array", "переопределение массива". Выход за границы памяти возникает, если для создания переменной данного размера не хватает места.

Если имя массива использовалось до выполнения оператора DIM (или без него), тогда к нему атоматически применяется оператор DIM с максимальными границами для каждого индекса, равными 10. Может существовать только один массив данным именем и типом и заданным числом размерностей (= < 255).

Использование: Кроме очевидного использования для создания массивов требуемого типа и размера, оператор стоит применять для объявления всех переменных. При существовании в программе больших массивов для создания новых простых переменных потребуется несколько секунд; определение же всех скалярных величин в начале программы позволяет сэкономить это время. Для ускорения работы программы можно сперва определить часто используемые простые переменные (особенно в больших циклах).

Заметьте, что любой массив можно уничтожить с помощью оператора ERASE и переопределить затем с помощью другого оператора DIM. DIM —выполняемый оператор; *он может находиться в любом месте программы, но лучше задавать его перед работой с массивом, к которому он относится.

Каждый созданный элемент принимает начальное значение, равное байту нулей (значение 0 или пустая строка).

Пример:

```
100 GOSUB 5000 ' инициализация
200 . . .
5000 DIM X, A, START, Z1, Z2, M (3,200), X% (1000)
5010 RETURN
```

См. также: Раздел главы 2 "Форматы хранения переменных".

Цель: Выполняет цепочку графических команд

Формат: DRAW *строковое выр.*

Строковое выр. :: = любое выражение, возвращающее результат строкового типа.

Действие: Эта команда использует "Графический макроязык" GML. С использованием GML можно быстро определить сложные наборы команд рисования для выполнения в любом из графических режимов. Более того, можно создавать строки "подпрограмм", которые могут вызываться командой языка GML.

"Программа" на GML состоит из строки длиной до 255 символов. Строку составляют наборы команд, использующие ключевые символы; Команды могут разделяться пробелами для удобства чтения. Некоторые команды должны завершаться точкой с запятой (;), в коде ASCII — (3B), во избежание путаницы. Каждая команда состоит из одной буквы (некоторые команды имеют необязательный префикс), за которой обычно следует один аргумент.
В большинстве случаев аргумент — это числовая константа, которую можно заменить ссылкой на переменную.

Ниже приводится краткий перечень команд и их особенности:

U *длина* :: = рисует вертикальную линию вверх от положения графического курсора*) длиной *длина* пикселей

D *длина* :: = рисует вертикальную линию вниз от графического курсора длиной *длина* пикселей

L *длина* :: = рисует горизонтельную линию влево от графического курсора на расстояние *длина* пикселей

R *длина* :: = рисует горизонтальную линию вправо от графического курсора на расстояние *длина* пикселей

E *длина* :: = рисует линию под углом 45° длины *длина*

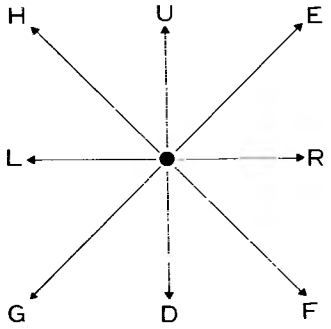
F *длина* :: = рисует линию под углом 315° длины *длина*

G *длина* :: = рисует линию под углом 225° длины *длина*

H *длина* :. = рисует линию под углом 135° длины *длина*

*) графический курсор — точка/пиксель на экране, являющаяся точкой отсчета для очередной операции GML

Рисунок отражает действие этих команд (все углы равны 45°):



удиление — *длина* пикселей от цетнральной точки

U D L R E F G H

Существуют дополнительные команды для проведения линии до точки с заданными относительными или абсолютными координатами.

M *по X, по Y* :: = перемещение от графического курсора до заданной точки. Если явно

задан знак (+ или -), то относитвльная координата отсчитывается от текущего положения графического курсора:

M ± расстояние по X, ± расстояние по Y

Типы координат — относительные и абсолютные — могут использоваться в команде M в любой комбинации.

B *команда* :: = движение без рисования следа

N команда :: = движение с рисованием следа т возврат графического курсора в исходное положение.

Эти команды могут также содержать следующие команды "состояния" (кроме абсолютных координат M):

A *вращ.* :: = поворот каждого заданного направления на 90°. Допустимые значения:
0 = нормальное направление
1 = поворот на 90° по часовой стрелке
2 = поворот на 180° по часовой стрелке
3 = поворот на 270° по часовой стрелке

S *масштаб* :: = умножить каждую длину на заданный коэффициент — масштаб может иметь значение от 1 до 255; при 0 и 4 умножение не производится. Козффицент масштабирования делится на 4 и умножается на каждую задаваемую длину. Таким образом S1 означает, что длины отрезков умножаются на 1/4, а S12 на 3.

Масштабирование действует на все, что рисуется без префикса B.

C *цвет* :: = установка по умолчанию нового цвета изображения;
0 = < *цвет* < = 15 (См. COLOR).

Графические *подпрограммы* можно выполнять следующим образом:

X *строковая-пер.*; :: = чтение текста, содержащегося в переменной *строковая-пер.*, с интерпретацией его как команды GML. Такие команды могут быть вложенными. Точка с запятой является признаком конца строки.

В любой из упомянутых выше команд, где требуется числовой аргумент, его можно заменять переменной:

= *имя-перем;*

Символ "—" и точка с запятой ограничивают имя переменной, содержащей аргумент со значением, трубуемым для этой команды, Переменная может быть злементом массива; в этом случае ее индекс также может быть переменной или константой — но не выражением. В команде M точка с запятой не заменяет запятую.

Примеры:

```
M= dl;„ - =hl;    U = ST (5) ;    C = SCOL ;
```

Наконец, заметим, что первый аргумент команды M — представляющий координату X, определяет, является ли команда абсолютной или относительной: если он имеет знак плюс или минус (+, —), вся команда M является относительной, иначе — абсолютной; знак (или его отсутствие) во втором аргументе не отменяет это определение.

Запомните, что при переходе в графический режим экрана графический курсор, масштаб, угол ориентации и цвет изображения остаются прежними.

Использование: Эта команда имеет очень много возможностей. Во-первых, она является самым компактным способом кодирования любой фигуры. Во-вторых, нарисованная фигура может быть увеличена (с помощью команды S) и повернута налево (направо) и вверх (вниз) (с помощью команды A).
В-третьих, она позволяет обрацаться к строкам, которые можно объединять и использовать для создания сложных рисунков (команда X).
Наконец, она имеет средства для установки графического курсора в любую точку зкрана и для сохранения его в этой позиции даже после проведения отрезка.

Отметим, что с помощью команды DRAW нельзя начертить линию под произвольным углом (так, как это делается в модели языка Лого); ее также нельзя использовать для работы в полярных координатах. Однако, это можно смоделировать с помощью комбинации:

```
310 DRAW "некоторая фигура"
320 X9 = RADIUS * SIN (ANGLE)
330 Y9 = RADIUS * COS (ANGLE)
340 DRAW "M = X9;„ = Y9;"
```

При этом чертится линия длины RADIUS в направлении ANGLE. Еще лучше использовать цикл с парами RADIUS/ANGLE.

Запомните, что графические команды работают в воображаемом пространстве с координатами -32768 = < *коорд* < = 32767. Однако, на экране будет видна только та часть текста, для которой 0 = < X < = 255 и 0 = < Y < = 191.

Можно создавать различные строки, представляющие в совокупности некоторый объект; затем с использованием команды X можно нарисовать окончательный рисунок. Этот прием увеличивает также максимально допустимую длину вашей команды.

Запомните, что командная строка, вызванная командой X, может сама содержать команды X. Это используется при создании нестандартного "алфавита" — каждый его символ определяется как набор команд DRAW в оператора DATA.

При этом допускается поворот и увеличение "символов". Можно, например, составить такую команду DRAW (предполагается, что массив C$ содержит строку DRAW для каждого символа с соответствующим индексом в кодовой таблице ASCII):

```
100 TITLE$ = "Заголовок": GOSUB 500
110 DRAW "BM20, 120 C9 A1 S2 Xds$;"
. . . остальная часть программы
500 DS$ = "  ": FOR X=1 TO LEN (TITLE$)
510 DS$ = DS$ + XC$ (ASC (MID$ (TITLE$, X)))
520 NEXT X: RETURN
```

Пример:

```
10 SCREEN 2
20 BOX$="R20U20L20D20":HAT$="E14F14BL20"
30 DO$="R5U10L5D10":WINDOW$="R4U4L4D4"
40 HOUSE$="XBOX$;BR4XDO$;BM10,-5XWINDOW$;BM-18,-12XHAT$;"
50 DRAW"BM20,20C1S2Xhouse$;BM100,100S6C15Xhouse$;"
60 REM рисует небольшой черный дом слева вверху; большой белый
   дом — в центре
70 GOTO 70
```

См. также: COLOR, LINE, CIRCLE, раздел "графическое программирование".

Цель: Сигнализирует о конце выполнения программы и закрывает все файлы.

Формат: END

Действие: Прекращает выполнение программы, и возобновляет прямой режим. Закрывает все открытые файлы (как функция CLOSE без параметров). Может находиться в любом месте программы.

Использование: Оператор выдает подсказку "Ok" — так же, как после выполнения строки с самым большим номером в программе. Однако, полезно бывает помещать его в середину программы. Например, если пакет подпрограмм начинается со строки 10000, то 9999 END предотвратит случайные обращения программы к подпрограммам, вызывающие ошибку. Другое применение — внутри условного оператора для быстрого выхода из него:

строка IF *условия* = 0 THEN END ELSE

является допустимой.

Наконец, END удобен для документации, обозначая несомненный физический конец программы.

Пример:

```
20 PRINT "ВСЕ, РЕБЯТА!": END
```

См. также: RUN, STOP, описание режимов работы в начале Гл. 3.

Цель: Указывает, что достигнут конец файла

Формат: *x* = EOF (*ном. файла*)

ном-файла :: = допустимое значение, указывающее номер открытого файла; должно быть меньше или равно MAXFILES, и 0 = < *ном-файла* < = 15.

Действие: Во-первых, проверяется, был ли открыт файл с заданным номером; если нет, выдается сообщение "File not OPEN", "файл не открыт". Если *ном-файла* больше MAXFILES (или больше 15), выдается сообщение "Bad file number", "неправильный номер файла". Потом идет проверка на содержание данных во входном буфере — если их нет, буфер заполняется. Наконец, проверяется, является ли следующий (непрочитанный) символ признаком EOF (это-Control Z, 1A). Если это так, то функция возвращает значение -1 (истина); в противном случае 0 (ложь). Если EOF ( ) возвращает значение TRUE (-1), то любая попытка ввести данные из файла приведет к сообщению "Input past end", "ввод эапрещен".

Эта функция имеет смысл при последовательном вводе. Однако, при последовательном выводе она возвращает нулевое значение (ложь), — НО! Использование функции EOF ( ) при выводе (особенно если не производилась запись данных) может привести к "бесконечной" работе магнитофона. В простейших комплектах магнитофон является единственным устройством, с которым работает функция EOF ( ), но каждое дополнительное устройство (особенно дисководы!) должно вести себя в ситуациях, описанных выше, идентичным образом.

Использование: Функция EOF ( ) позволяет работать в условиях, когда неизвестно количество данных в файле; благодаря ей даже случайно нельзя вызвать ошибку "ввод запрещен". Функцию EOF ( ) можно проверить на эту ошибку перед оператором ввода. Она возвращает логическое значение и может быть использована в операторе IF/THEN. См. пример.

Надо следить за тем, чтобы файл находился в режиме INPUT; проверка здесь не производится, и результат может быть опасным.

Пример:

```
10 OPEN "cas:data" FOR INPUT AS #1
20 IF EOF (1) THEN 40
30 LINE INPUT #1, A$: GOSUB 10000: GOTO 20
40 CLOSE 1: PRINT "You have processed all the data!"
10000 REM — подпрограмма обработки входной строки
```

См. также: CLOAD, CSAVE, LOAD, SAVE, раздел "ввод-вывод файлов".

Цель: Уничтожает массив в памяти, что позволяет повторно использовать область ОЗУ и имя массива

Формат: ERASE *имя-масс* [, *имя-масс*] . . .

имя-масс :: = имя любого существующего массива без круглых скобок

Действие: При выполнении этого оператора массива все данные, содержащиеся в них, теряются. Эти имена массивов могут в последующем быть заново объявлены и использованы так, будто они раньше никогда не использовались. Освобожденная область памяти добавляется к рабочей области, размер которой дает функция FRE( ).

Если какая-либо переменная из списка массива не является активной переменной массива, выдается сообщение: "Illegal function call", "неправлильный вызов функции". Если случайно используется имя массива со скобками — это синтаксическая ошибка. Но если попытаться объявить заново с помощью оператора DIM имя массива без оператора ERASE выдается сообщение "Redimensioned array", "переопределение массива".

Использование: Во-первых, оператор ERASE освобождает занятие области памяти. Если для инициализации ввода-вывода или других действий требуется много памяти, то сначала заведите для этих целей массивы, а после окончания этих действий примените ERASE.

Другой пример использования — регулирование границ массива с учетом выполняемой функции. В предположении, что имеется достаточно места, можно создать временный массив для хранения старых значений, затем сам массив уничтожается

оператором ERASE, а его имя опять используется в операторе DIM с новыми границами; данные копируются в новый массив из временного массива, который затем уничтожается для освобождения места. Для временного хранения, если памяти не хватает, можно использовать видео-память.

При нехватке памяти ERASE является единственным средством для управления использованием ОЗУ. Запомните: в любом месте, где имя массива используется первый раз, если это не оператор DIM, используется неявный оператор DIM с границей каждого индекса, равной 10. Поэтому если массив уничтожается, следует в этом месте применить оператор DIM, чтобы случайно не присвоить новому массиву неверный размер.

Пример:

```
10 DIM A (100,20): FOR X=1 TO 100: FOR Y=1 TO 20
20 INPUT A (X,Y): NEXT Y,X
30 GOSUB 10000 ' обработка данных
40 ERASE A ' область памяти восстановлена, продолжайте . . .
```

См. также: DIM, FRE ( ) , Приложение "Карта памяти", раздел "Форматы переменных".

Цель: Возвращает номер последней строки, в которой произошла ошибка

Формат: *x* = ERL

Действие: Функция возвращает номер последней строки, в которой произошла ошибка, т.е. для которой выдалось бы сообщение "XX ошибка в XXXX".

Возвращаемое значение является 16-битовым целым числом без знака, хранящимся в слове ERRLIN (F6B3) и при загрузке принимающим нулевое значение. Любая ошибка в прямом режиме работы программы устанавливает его равным номеру строки, содержащей ошибку. Больше ничто не может повлиять на это значение. Такие же присваивания происходят в операторе ERROR.

Использование: Применяется в основном для подпрограмм обработки ошибок, выбираемых с помощью оператора ON ERROR GOTO. Позволяет определить причину ошибки; одна программа обработки обычно охватывает несколько типов ожидаемых ошибок.

Заметьте, что хотя ERL является функцией, после ее имени нельзя использовать круглые скобки; ERL (0) является недопустимым выражением.

Пример:

```
10 ON ERROR GOTO 100
20 A= B / C
30 Z= CINT (A)
. . .
100 IF ERL= 20 THEN C= 1E—10: RESUME
110 IF ERL= 30 THEN A= 32767: RESUME
120 PRINT "Error"; ERR; "in line"; ERL: STOP
```

См. также: ON ERROR, ERR, ERROR

Цель: Возвращает номер последней зафиксированной ошибки

Формат: *x* = ERR

Действие: Функция возвращает номер последней обнаруженной ошибки. В любом месте, где происходит ошибка, ее номер запоминается. На функцию ERR может оказывать воздействие только оператор ERROR.

Использование: ERR применяется в программах обработки ошибок. Если в одном операторе возможно возникновение сразу нескольких различных ошибок, то ERR применяется для обращения к различным программам обработки с помощью оператора ON ERROR.

Если нужное сообщение об ошибке исчезло с экрана (например, после команды LIST), а оно еще нужно, то оператор ERROR ERR восстановит его. Полный список возможных сообщений и их номеров см. в приложении. Хотя ERR и функция, после нее нельзя ставить круглые скобки: ERR (0) недопустимо.

Пример:

```
10 ON ERROR GOTO 100
20 X = CINT (X/Y)
30 . . .
100 IF ERR= 11 THEN GOSUB 200: RESUME ' divide by zero
110 IF ERR=6 THEN GOSUB 300: RESUME' Overflow
120 PRINT "Error"; ERR; "in line"; ERL: STOP
```

См. также: ERL, ERROR, ON ERROR GOTO

Цель: Искусственный способ имитировать ошибку

Формат: ERROR *ном-ошиб*

ном-ошиб :: = выражение, значением которого является целое число от 0 до 255

Действие: Оператор ERROR имитирует возникновение ошибки. Аргумент *ном-ошиб* является номером сообщения об ошибке (см. таблицу сообщений и их номеров в приложении). При этом заданная номером ошибка возникает в том месте, где стоит оператор ERROR. Перехват ошибки, ERR и ERL работают так, словно в данном месте действительно произошла ошибка.

Если аргумент больше 255 или меньше 0, выдается сообщение: "Unprintable error", "неизвестная ошибка", но функция ERR будет возвращать точный номер ошибки. Этот оператор всегда имитирует в программе состояние ошибки.

Использоапние: Во-первых, если вы забыли последнее сообщение, то набрав команду ERROR ERR, можно снова его прочитать (или, например, требуется быстро узнать, что такое ошибка 12). Во-вторых, этот оператор используется как один из способов ветвления: можно направить выполнение программы в выбранную точку с помощью одной из подпрограмм в ON ERROR GOTO, которая затем использует ERR для выбора одного из путей ветвления.

Наконец, ERROR используется для восстановления исходного сообщения об ошибке в программе, вызываемой по ON ERROR, если других возможностей восстановления нет.

Пример:

```
10 ON ERROR GOTO 10000
20 ERROR 255
25 A=B/0
...
10000 IF ERR = 255 THEN GOSUB 11000: RESUME: NEXT
...     ' обработка всех восстанавливаемых ошибок
10500 PRINT "Строка";  ERL;":";: ERROR ERR: END
```

См. также: ERR, ERL, ON ERROR GOTO, RESUME, приложение "Сообщения об ошибках".

EXP

фукция

Цель: Возвращает значение *e* в степени x

Формат: *x* = EXP (*экспонента*)

экспонента :: = африфметическое выражение:
−149.66803104461 = < *экспонента*
< = 145.06286085862

Действие: Вычисляет значение e в степени, заданной аргументом. Значение e берется равным 2.7182818284588. Это несколько неточно, т.к. на самом деле e = 2.7182818284590 — допускается погрешность 2 в последней значащей цифре. Кроме того, поскольку, экспоненциальные значения быстро растут, аргумент должен быть относительно небольшим. В противном случае выдается сообщение о переполнении.

Использование: В основном применяется в научных расчетах. Константа e может быть задана с помощью EXP (1). Эта функция является обратной к логарифмической функции LOG ( ), которая вычисляется с помощью натурального логарифма. Таким образом, EXP (LOG (X)) дает в результате X. Заметим, что в последней значащей цифре здесь допускается погрешность.

Пример:

```
PRINT LOG (EXP (3) )
2.9999999999998
Ok
```

См. также: LOG ( )

FIX

Цель: Возвращает целую часть аргумента

Формат: *x* = FIX (*ар-выр*)

ар-выр :: = любое арифметическое выражение

Действие: Функция отбрасывает часть результата *ар-выр* справа от десятичной точки, возвращая целую часть вычисленного аргумента.

Она не похожа на функцию INT ( ), которая округляет в сторону меньшего целого; FIX ( ) вообще не округляет цифр. Однако, для положительных аргументов FIX ( ) и INT ( ) идентичны.

Использование: Функция удобна для форматного вывода. Она может также использоваться для определения делимости: IF x/n = FIX (x/n) THEN → x делится на n нацело.

В большинстве случаев, если значение надо преобразовать к целому числу, можно использовать FIX ( ), но в большинстве программ используется INT ( ). FIX ( ) на 2% медленнее, чем INT ( ).

Функцию FIX ( ) можно использовать для округления:

```
x = FIX (n*100) / 100 ' не округляет
x = FIX (n*100+.5) / 100 ' округляет по правилу 5/4
```

См. также: INT ( )

Цель: выполняет с повторением группу операторов

Формат: FOR *счетчик* = *начало* TO *конец* [STEP инкремент]

счетчик :: = арифметическая переменная, хранящая текущее значение счетчика.
начало :: = арифметическое выражение, задающее начальное значение счетчика.
конец :: = арифметическое выражение, задающее конечное значение счетчика.
инкремент :: = арифметическое выражение, задающее величину, прибавляемую к четчику при каждом повторении цикла; по умолчанию инкремент равен 1

Действие: этот оператор широко используется в Бейсик-программах. При выполнении оператора FOR... первый раз вычисляется значение выражения "*начало*", которое присваивается переменной "*счетчик*". Затем идет обычное выполнение операторов. При достижении парного оператора NEXT переменная "*счетик*" увеличивается на значение, заданное выражением "*инкремент*" (или уменьшается, если *инкремент* отрицателен). Значение счетчика сравнивается со значением выражения "*конец*" — если счетчик больше, выполнение нормально продолжается после оператора NEXT; если нет, то управление передается обратно оператору FOR..., и снова выполняются все операторы между FOR и NEXT. Заметим, что при отрицательном значении инкременте идет уменьшение счетчика и в этом случае *счетчик* должен быть меньше значения "*конец*" для перехода к операторам, следующим оператора NEXT.

Каждый оператор FOR занимает при работе 25 байтов стека (свободного места в памяти), не считая места, отводимого под переменную

"*счетчик*" (стек освобождается только при завершении цикла, после выполнения оператора NEXT с условием конца). Операторы FOR могут быть вложенными, в том смысле, что самый внутренний цикл выполняется первым — основным правилом является различие переменных "*счетчик*", а также то, что для самого внешнего цикла оператор NEXT не выполняется до тех пор, пока не выполнится NEXT, относящийся к внутреннему циклу. Это означает, что оператор NEXT для внутреннего цикла должен стоять перед соответствующим NEXT внешнего цикла.

Парный оператор NEXT может не использовать переменную "*счетчик*" (это на 20% быстрее). Если требуется, чтобы несколько вложенных операторов FOR заканчивались в одном месте, можно использовать один оператор NEXT, содержащий список переменных "*счетчик*", разделенных запятыми; первое имя в списке соответствует ее внутреннему, последнему — внешнему циклу. (см. *NEXT*).
В отличие от рекомендаций стандарта ANSII, (или правила, принятого в других языках), оператор FOR и операторы тела цикла всегда выполняются один раз, даже если условие "*конец*" истинно в самом начале.

Заметьте, что выражения "*начало*", "*инкремент*" вычисляются при первом выполнении операторa FOR и заносятся в стек; эти значения нельзя менять внутри цикла. Однако значение переменной "*счетчик*" может меняться в цикле, и цикл заканчивается только, когда значение "*счетчик*" будет больше значения "*конец*". Поскольку указатель на оператор, следующий после FOR, помещается в стек, не следует использовать самоизменяющийся код в теле цикла, иначе при выполнении NEXT возникнет ошибка.

Примеры вложенных циклов:

```
10 FOR X = 1 TO 100
20 FOR Y = 1 TO 10
30 PRINT "Этот оператор будет выполнен 1000 раз"
40 NEXT Y
50 PRINT "А этот — только 100 раз"
60 NEXT X
```

```
10 FOR X=1 TO 7: FOR Y=1 TO 3: READ A (X, Y): NEXT Y, X
```

```
10 FOR X=1 TO 10 STEP .1
20 IF FNA (X) = A (W) THEN X=10: NEXT:
GOTO 100
30 PRINT "Продолжаем поиск..."
40 NEXT
50 PRINT "Не нашли!": STOP
100 PRINT "Нашли!"
```

Использование: оператор FOR/NEXT часто используется в Бейсик-программах. Он используется для повторения участков программы. Эта возможность используется для повторения одного оператора или для того, чтобы изменит логику программы, использующей "переменную цикла" *счетчик* в качестве основы для выбора порядка выполнения операторов. Эта возможность позволяет включить весь текст основной программы в цикл — например, для выполнения итераций. Нельзя забывать, что этот оператор — единственный способ в MSX-Бейсике для изменения прямого порядка выполнения операторов без задания номеров строк, т.е. это структурный оператор.

Циклом FOR можно описать цикл WHILE. Следующая последовательность операторов заменяет цикл WHILE условие ... WEND:

```
10 IF NOT CND THEN 60 ' условие ложно
20 FOR X = -1 TO 0
30 ' оценка условия
40 X = CND ' если условие ложно, то цикл заканчивается
50 NEXT
60 ' оставшаяся часть ярограммы
```

Если здесь использовать STEP 0, цикл будет бесконечным. Гарантируется, что при ложном условии цикл вообще не будет выполняться. Пользователи языка Паскаль могут считать цикл FOR/NEXT разновидностью структуры REPEAT/UNTIL (со встроенным инкрементом).

Последнее замечание: цикл FOR/NEXT интересно используется в вызове подпрограмм (см. GOSUB)

См. также: GOSUB

Цель: сообщает объем свободной оперативной памяти; может начать "сборку мусора"

Формат: *x* = FRE (*фиктивный аргумент*)

фиктивный аргумент :: = любое имя строковогс или числового типа — важно не значение аргумента, а его тип.

Действие: Если фиктивный аргумент — числового типа, то подсчитывается и сообщается объем свободного пространства между дном стека и верхушкой области переменных. Если тип оргумента — строковый, то область, отведенная под строки, реорганизуется путем "сборки мусора", а затем сообщается размер свободного строкового пространства.

Для получения размера свободной области переменных вполне достаточно вызвать функцию следующим образом; FRE (0). Свободное пространство, обьем которого определен функцией FRE ( ), может быть использовано для стека процессора Z-80 во время выполнения программы (надо учитывать, что это всегда вызывает некоторую задержку в работе), для создания новых переменных или для вводимого нового текста программы на языке Бейсик.

С другой стороны, обращение FRE (" ") рассматривается как вызов функции со строковым аргументом. Возвращается значение, не превышающее размера пространства, зарезервированного оператором CLEAR n (по умолчанию 200 байт по включении). Части строкового пространства, отведенного под строковые данные, сдвигаются вниэ; эта операция производится автоматически, когда в программе большая часть строкового пространства уже использована в результате выполнения строковых операций. Эта операция занимает значительное время.

Периодическое задание FRE ( ) позволит проводить сборку мусора в удобное время. Следует заметить, что некоторые строки могут храниться в самой программе и, таким образом, не занимают строкового пространства.

Поскольку система MSX—Бейсик может использовать только 32К свободной памяти, FRE ( ) возвращает значение от 0 до 29282 (при максимальном обьеме памяти, CLEAR 0 и MAXFILES = 0). Кроме того, FRE (0) выдает сообщение "Out of memory", "Не хватило памяти" при достижении значения, меньшего 130, байтов, минимально допустимого для стека системы MSX—Бейсик.

Использование: Этот оператор выполняет много различных функций. Во-первых, он может использоваться для определения обьема свободной памяти (которая может составлять от нескольких сотен байт до 29К, т.к. часть диапазона адресов памяти могут использовать кассеты ПЗУ). Если вам нужно использовать этот параметр (или результаты какого-либо вычисления на его основе) в дальнейшем для CLEAR или DIM, то следует поместить этот значение в каком-либо "безопасном месте" таком, как область SLTWRK (FD09), где первые четыре байта отведены исключительно для использования программами в ПЗУ системы MSX-Бейсик, но не используются ими.

Другое обычное использование FRE ( ) — это заблаговременное предупреждение о возможной нехватке памяти. Поскульку Бейсик-система занимает часть стекового пространства для интерпретации вашей программы. вы должны держать свободной область объемом 130 байт во избежание ошибок, связанных с распределением памяти. Если такая ошибка произойдет, то вам придется немедленно освободить

некоторую область памяти, чтобы предотвратить зацикливание. Эффективным методом для этого служит определение небольшого массива; тогда первым шагом в вашей процедуре обработки ошибки должно быть выполнение оператора ERASE для этого массива, высвобождающего пространство для попытки восстановления.

Еще одно использование строкового варианта FRE (" "): "сборка мусора". Вопреки общепринятому мнению "сборка мусора" не ускоряется от частого использования; скорость ее зависит исключительно от количества задействованных строковых элементов и особенно строковых массивов. (Большой, но не слишком, строковый массив может послужить причиной 5-ти минутной задержки!). При выполнении этой операции система полностью "застывает" до окончания процесса. Для избежания этого следует либо не использовать слишком много строковых операций (записывая данные в область памяти, недоступной Бейсик-системе, с помощью операторов PEEK ( ), POKE, VPEEK ( ) или VPOKE), либо использовать "локальные" строковые функции (MID$( ), SWAP, LSET и RSET) вместо строковых пересылок. В любом случае эта функция позволяет организовать "сборку мусора" в точке, где, скажем, вы должны предупредить пользователя о наступающей задержке, а также позволяет отсрочить "сборку мусора", если зарезервировано достаточное строковое пространство. (Если ваше строковое пространство лишь немногим больше обычно используемого вами, вы будете все время "собирать мусор"; советуем оставить небольшое свободное пространство для строковых операций). Наконец, FRE (" ") позволяет вам контролировать, насколько заполнено строковое пространство, выбранное с помощью CLEAR

размер и соответственно подобрать этот параметр.

Функция FRE ( ) может помочь вам также в защите программы. Проверьте в определенном месте, соответствует ли свободное пространство отведенному начально, или если нет, принимайте меры; это удержит кого бы то ни было от модификации вашей программы или ее переменных. Разумеется, вам следует помнить, что определенное количество переменных было создано в этом месте.

Кстати, хотя параметры FRE ( ) игнорируются, и важен только их тип, выражение, однако, оценивается: если упоминается имя новой переменной, то она не создается, не уменьшая тем самым свободное пространство. FRE ( ) не может стать причиной выдачи в этом месте сообщения "Out of memory", "не хватило памяти".

Пример:

```
10 IF FRE (0) > 20000 THEN 30
20 PRINT "Эта программа требует 32K": STOP
30 LIMIT = (FRE (0)—12000) + 220 ' рабочие ячейки
40 POKE & HFD09, LIMIT: CLEAR LIMIT*1.2: LIMIT=PEEK (&HFD09)
50 DIM DUMMY (10), AA (LIMIT, 3), . . .
60 ON ERROR GOTO 1000
. . .
1000 ERASE DUMMY
1010 IF ERR < >7 THEN DIM DUMMY (10): GOTO 1100
1020 . . . ' программа перераспределения памяти
1010 . . . ' процедуры исправления других ошибок
```

См. также: DIM, CLEAR, Приложение "карта памяти".

Цель: Передает управление подпрограмме с возвратом на следующий оператор.

Формат: GOSUB *номер строки*

номер строки :: = номер строки в программе:
0 = < *номер строки* < = 65529

Действие: GOSUB используется для вызова подпрограммы — группы операторов, начинающейся со строки с номером *номер строки* и заканчивающейся оператором RETURN. Эта группа операторов может находиться в любом месте вашей программы. Главной особенностью GOSUB является то, что независимо от того, сколько раз, откуда и в каком порядке вы выполняете его, оператор RETURN возвращает управление оператору, стоящему за последним выполненным GOSUB.

В этом смысле оператор GOSUB похож на оператор GOTO, но он помещает в стек 7 байтов, содержащих информацию о том, куда возвращать управление. Можно рассматривать оператор GOSUB как средство расширения языка. Если вы должны выполнить сложную операцию — например, чтение файла данных в массив — вы можете поместить оператор GOSUB в том месте, где вы хотите выполнить эту задачу; позднее вы можете разработать специальную подпрограмму, осуществляющую это действие. Подпрограмма может начинаться с комментария.

Следует помнить, как осуществляется поиск заданной строки в тексте Бейсик-системой. Либо вся программа просматривается от GOSUB до конца, либо от начала до GOSUB. Соответственно, помещайте ваши подпрограммы либо в самом начале программы, либо сразу после GOSUB.

Однако, поиск осуществляется очень быстро, поэтому все вышесказанное употребимо для наиболее критических программ. Точка (.) вместо номера строки является недопустимой и вызывает сообщение о синтаксической ошибке, а использование несуществующего номера строки вызывает появление сообщения "Undefined line number", "Не определен номер строки".

Использование: Оператор GOSUB является основным инструментом структурного программирования. Методом нисходящего проектирования вы можете структурировать главную программу, используя GOSUB для вызова каждого сложного фрагмента. В качестве подпрограммы вы можете ввести "заглушку" типа PRINT "Вот здесь.": RETURN; это удобно, для отладки. Затем, будучи уверенным в правильности логического построения программы, вы детально проектируете и "расписываете" каждую подпрограмму, вызывая другие подпрограммы по мере необходимости. Эта последувательная декомпозиция продолжается, пока программа не будет завершена и проверена.

В другом методе — "снизу-вверх" — вы сначала пишете основные подпрограммы низкого уровня; проверив их, вы добавляете подпрограммы более высокого уровня, которые вызывают подпрограммы низкого уровня, и так до тех пор, пока не достигнете верхнего уровня. Возможно, что лучшим является гибрид двух методов; метод "снизу-вверх" пригоден для больших библиотек стандартных, проверенных процедур. В обоих случаях каждая под программа должна быть небольшой так, чтобы можно было охватить одним взглядом всю ее логику. Вы должны быть особо внимательны при нумерации строк.

Наиболее распространенной ошибкой при использовании GOSUB является неполное отделение подпрограммы от главной программы. При случайном попадании в подпрограмму без использования GOSUB она будет работать нормально, но выдается сообщение об ошибке "RETURN without GOSUB", "RETURN без GOSUB".

Другой сложностью является использование циклов FOR/NEXT. Внутри цикла можно использовать GOSUB, но оператор NEXT должен быть вызван только после оператора RETURN. В противном случае возникнет сообщение "Next without FOR", "NEXT без FOR":
С другой стороны, если цикл FOR используется внутри подпрограммы, то RETURN в контексте активного FOR прерывает действие цикла. Это наводит на мысль, что наиболее эффективно размещать программу поиска массива внутри подпрограммы. Как только элемент найден, можно выполнить выход из подпрограммы.

Немного о рекурсии. Поскольку стек Z-80 используется для самых разных целей, вызовы подпрограмм являются рекурсивными, т.е. они могут вызывать сами себя (при условии, что они "знают", когда остановиться прежде, чем память полностью использована) . Однако, переменные во всех подпрограммах — везде, кроме определенных функций — являются глобальными и статическими. Это означает, что они созданы явно, существуют постоянно и все в равной степени доступны из любой точки. Следовательно, важно определить конкретные имена переменных для каждой подпрограммы и использовать их для передачи параметров и возвращения результатов. Умело используя массив для каждой локальной скалярной переменной, можно змулировать рекурсивную подпрограмму:


```
10 DIM A (48): A = 0 ' локальные переменные для под-
   программы FACT
20 INPUT "FACTOR (0–48)"; F1
30 GOSUB 1000: PRINT "факториал равен" ; F0: STOP
1000 ' Подпрограмма FACT — вход F1, результат F0
1010 A = A+1: A(A) = F1
1030 IF A(A) = 1 THEN F0 = 1: GOTO 1050
1040 F1 = A(A) – 1: GOSUB 1000: F0 = F0*A(A)
1050 A = A – 1: RETURN
```

Этот прием нужен для того, чтобы полностью изолировать каждую локальную переменную и сохранить все переменные, увеличивая счетчик каждый раз при входе в подпрограмму и уменьшая его при выходе. Конечно, любая рекурсивная программа может быть написана как цикл, но многим больше нравится этот метод. Кроме того, действительно хорошей практикой является изолировать все входы и выходы в программе и иметь одну входную и одну выходную точку. Это необязательно, но диктуется здравым смыслом.

И, наконец, оператор GOSUB может оказаться полезным при использовании в качестве команды прямого режима.

Если ваша программа состоит из функциональных подпрограмм, которые вы хотите проверить, то, используя команду GOSUB в прямом режиме, можно войти в эту подпрограмму. При этом в последующих операторах GOSUB и GOTO могут задаваться любые номера строк: эта команда переводит вас в режим RUN. Когда выполнится самый внешний оператор RETURN вы увидите, что система MSX-Бейсик вернется в прямой режим, в котором вы задали данную команду.

Пример:

```
10 I=0: GOSUB 100
20 I=1: GOSUB 100
30 I=2: ON I GOSUB 50, 100
40 END
50 RETURN ' фиктивная подпрограмма
100 ' ПОДПРОГРАММА
110 PRINT "ВЫЗОВ"; I
120 RETURN
```

См. также: RETURN, GOTO

Цель: Передает управление оператору с заданным номером строки.

Формат:

GOTO *номер строки*
или
GO TO *номер строки*

номер строки :: = любой существующий номер строки в программе
0 = < *номер строки* < = 65529

Дейсивие: Оператор GOTO непосредственно передает управление другому оператору вашей программы. Если в программе нет строки с заданным номером, выдается сообщение "Undefined line number", "номер строки не определен". Управление передается заданной строке, даже если в ней нет выполняемого оператора; например, это может быть оператор REM. В случае, если оператор записывается двумя словами GO TO, то он выполняется аналогично — этот формат сохранен совместимости с другими языками.

Использование: Этот оператор используется для связи отдельных частей вашей программы. Вообще следует по возможности избежать использования оператора GOTO, поскольку излишне частое использование его приводит к созданию программы, выглядящей очень запутанно.

```
100 ON VAL (INPUT$ (1)) GOTO 110, 120, 130
105 PRINT 0: GOTO 140 ' многопутное ветвление
110 PRINT 1: GOTO 140
120 PRINT 2: GOTO 140
130 PRINT 3
140 ' конец ветвления
```

Конечно, имеется много других применений GOTO. Любая хорошая программа должна использовать GOTO как можно меньше и только там, где это логически необходимо; для связок старайтесь использовать GOSUB.

Пример:

```
10 LIMIT = 10
20 FOR I=1 TO 10: NAM$ (I) = CHR$ (47+I) : NEXT
30 INPUT ID$
40 FOR X=1 TO LIMIT ' поиск ID$ в NAM$ (   )
50 IF ID$ = NAM$ (X) THEN X=LIMIT: NEXT: GOTO 100
60 NEXT
70 ' не найдено
80 PRINT "не найдено"
90 GOTO 30 ' назад к началу программы следующим значением ID$
100 ' обработка найденного значения
190 PRINT "найдено"
200 GOTO 30 ' назад к началу программы за следующим значением ID$
```

См. также: GOSUB, ON GOTO

Цель: Преобразование числового аргумента в шестнадцатеричную символьную строку.

Формат: *x$* = HEX$ (*арифм. выраж.*)

арифм. выраж. :: = любое арифметическое выражение, значение которого является целым числом со знаком или без него
-32768 = < *арифм. выраж.* < = 65535

Действие: Эта функция перекодирует аргумент, преобразуя результат в целое число и далее в символьную строку. Если аргумент выходит за указанный диапазон, то выдается сообщение об ошибке переполнения. Правильный аргумент преобразуется в символьную строку, байты которой соответствуют значению *арифм. выраж.*, представленному в шестнадцатеричном коде. Так, же как в функции STR$ ( ) и команде PRINT, ведущий знак и пробелы и завершающий пробелы не формируются. Более того, ведущие нули отсекаются. Это означает, что аргумент 257 даст "101" — другими словами, трехсимвольную строку, имеющую значение в коде ASCII (31), (30) и (31), или 49, 48 и 49 в десятичном виде. Следовательно, максимальная длина результата функции равна 4 байтам.

Если аргумент находится между нулем и 65535, то преобразование осуществляется с помощью беззнаковых оснований: HEX$ (0) → "0" и HEX$ (65535) → "FFFF". Однако, отрицательные значения от -1 до -32768 рассматриваются в дополнительном коде (как, собственно, Бейсик-система и работает с ними во внутреннем представлении) : HEX$ (-1) → "FFFF" и HEX$ (-32768) → "8000". Как ни странно, но HEX$ (32768) → также "8000". Для отрицательных аргументов HEX$ (*отр. арг.*) = HEX$ (65536 + *отр. арг.*). Очевидно, для того, чтобы ограничиться

использованием 16-битовых величин в дополнительном коде, надо выявлять и отбрасывать аргументы > 32767.

Функция VAL ( ) дает обратное преобразование:

```
in = VAL ("&H" + HEX$ (in))
```

Использование: Во-первых, эта функция удобна для преобразования текста при выводе, или просто преобразования целых чисел в шестнадцатеричные. Существование этой функции делает возможной шестнадцатеричную арифметику. Единственная проблема состоит в добавлении ведущих нулей для малых значений. Попробуйте такой прием:

```
DEF FN HX$ (in) = RIGHT$ ("0000" + HEX$ (in), 4)
```

Пример:

```
10 PRINT "16-ЧНАЯ АРИФМЕТИКА: ВВЕДИТЕ ДВА ЧИСЛА"
20 INPUT A$, B$
30 SM = VAL ("&H" + A$) + VAL ("&H" + B$)
40 DF = VAL ("&H" + A$) - VAL ("&H" + B$)
50 PRINT "Сумма, разность:"; HEX$ (SM); HEX$ (DF)
```

См. также: BIN$ ( ), OCT$ ( ), STR$ ( ), VAL$ ( )

Цель: Выбор одного из двух случаев на основе проверки условия.

Формат: IF *условное выражение* THEN *список операторов* [ELSE *список операторов*]

условное выражение :: = любое арифметическое выражение: нулевой результат = FALSE, все остальное = TRUE

список операторов :: = Несколько (или ни одного) операторов Бейсика, разделенных двоеточиями, или любой номер существующей строки.

Примечание: весь оператор должен быть расположен на одной строке и заканчиваться физическим признаком конца строки. Кроме того, вместо ключевого слова THEN может быть использовано GOTO (Прим. пер. — При этом за GOTO должен следовать номер строки).

Действие: Вычисляется значение условного выражения. Если выражение строковое, то выдается сообщение об ошибке "Type mismatch", "Несоответствие типов". Если результат равен нулю, то окончательный результат равен FALSE, в противном случае—TRUE. См. главу 2, в которой подробно описано вычисление выражений.

Если условие истинно (TRUE), то управление передается фразе THEN; если — FALSE и есть фраза ELSE, то управление передается фразе ELSE; если FALSE и нет фразы ELSE, то управление передается следующей строке программы. Если непосредственно за THEN или ELSE следует номер строки и управление передается этой фразе, то выполняется фактически оператор GOTO *номер*.

После ключевых слов THEN и ELSE может следовать любое число операторов (фраза); когда

управление передается любой из этих двух групп, все следующие операторы — либо между THEN и ELSE, либо между ELSE и концом строки — выполняются последовательно, как обычно. Любая из этих групп может содержать оператор IF/THEN/ELSE.

Оператор IF/THEN/ELSE занимает целиком нумерованную строку, от первого IF до физического конца строки. Никакой оператор, стоящий в строке за простым оператором IF *условное выр.* THEN *номер строки*, никогда не может быть выполнен. Проблема "болтающегося ELSE "рещается следующим образом: любой ELSE в строке связывается с ближайшим предшествующим ему в строке IF/THEN, которому еще не поставлен в соответствие ELSE, а не с первым (внешним) IF/THEN на строке (как в случае скобок). Пользуйтесь простым приемом: добавляйте нужное количество ELSE, начиная с конца строки для удовлетворения всех вхождений IF/THEN: строка оператора

```
IF FALSE THEN IF TRUE THEN PRINT 2 ELSE PRINT 3
```

при выполнении ничего не напечатает: группа ELSE, соответствующая первому IF/THEN, подразумевается в конце оператора. Для ясности можете добавить другие ELSE без последующих операторов: при выполнении строки

```
IF FALSE THEN IF TRUE THEN PRINT ELSE ELSE
PRINT 3
```

будет напечатано 3. THEN/ELSE можно воспринимать как пару скобок, добавленных в нужном количестве в конец строки, чтобы все было нормально.

Сложность с оператором IF заключается в том, что он занимает всю физическую строку. Если вы привыкли помещать на строке как можно больше операторов, то для вас жизненно важно быть внимательным к каждой строке, содержащей IF — все до IF выполняется как обычно; все после IF всегда является частью этого оперетора IF. Это обычный источник проблем — вы жестко ограничены тем, что на одной строке может быть только 255 символов.

Использование: Этот оператор является основным способом выбора пути в программе. Как основной условный оператор, IF/THEN широко используется в любой программе (помните, что операторы ON осуществляют выбор так же, как условные операторы). Любое арифметическое выражение может быть использовано как условие; при этом следует помнить, что если результат его вычисления равен нулю, то это FALSE во всех остальных случаях — TRUE. Например,

```
IF EOF (1) AND X3=5 THEN . . .
```

— совершенно правомерно; это означает, что, если X3 = 5 и вы находитесь в конце файла №1 (и EOF возвращает -1), то будут выполнены следующие операторы. Существенно, что вы можете использовать любые арифметические переменные в качестве логических значений, сохраняя условия для последующей проверки оператором IF. Это означает, что IF X< >0 THEN . . . идентично записи IF X THEN . . .

Конструкция ELSE IF во вложенных операторах IF совершенно ясна. Проверьте следующее:

```
IF ch = 13 THEN PRINT "возврат" ELSE IF ch = 32
THEN PRINT "пробей" ELSE IF ch = 48 THEN
PRINT "0" ELSE RPINT "неверная клавиша"
```

См. примеры

IF / THEN / ELSE

оператор

Пример:

```
10 IF FRE (0) < 10000 THEN PRINT "Недостаточно памяти": STOP
20 IF FRE (0) < 20000 THEN LIMIT = 300 ELSE LIMIT = 600
30 DIM  S (LIMIT) ' нельзя записать в конце предыдущей строки
40 PRINT "Нажмите любую клавишу по готовности"
50 IF NOT LEN (INKEY$) GOTO 50 ELSE PRINT "Привет"
60 GOSUB 100: IF RV >3 AND C1 THEN GOSUB 200 ELSE 90
70 IF C2 THEN GOSUB 300: GOTO 90
80 PRINT "Ошибка во втором условии": GOTO 60
90 IF RV > 3 THEN 60 ELSE PRINT "Пока!": END
.
.  Остальная часть программы
.
```

См. также: ON/GOTO и ON/GOSUB, раздел главы 2 "Логические операции"

фукция

INKEY$

Цель: Осуществляет ввод одного символа с клавиатуры.

Формат: *x$* = INKEY$

Действие: Функция пытается прочесть символ из буфера клавиатуры. Если символ отсутствует, INKEY$ возвращает пустую строку (" "). Если символ есть, он берется из буфера и передается в строковую переменную. Нажатие любой клавиши формирует код, который может быть прочитан этой функцией.

Эта функция может быть использована в любом месте, где ожидается строковый символ. Следует следить за двумя ситуациями: за возвратом пустой строки, что вызывает ошибку в нескольких строковых функциях; кроме того, клавиша STOP не читается, а Control-STOP чистается не всегда. Вообще эта функция является основным способом ввода с клавиатуры.

Главным преимуществом этой функции является то, что она не воспроизводит на экране символ прочитанного кода, не ожидает символа, если клавиша не нажата, и может читать любую клавишу (исключая STOP). Курсор не выводится на экран (пока не введен режим STOP нажатием соответствующей клавиши). Помните, что хотя INKEY$ может возвращать пустую строку, пока клавиша не нажата, в момент, когда она нажата, будет получен символ, который будет потерян, если его сразу не сохранить.

INKEY$ использует стандартный 40-символьный буфер клавиатуры KEYBUF (FBF0), и его указатели PUTPNT (F3F8) и GETPNT (F3FA) для получения символов с клавиатуры; Бейсик-система читает и буферзует их.

Использование: Наиболее общим случаем использования INKEY$ является ввод символа без использования стандартного режима INPUT; вы можете преобразовать символ перед выводом на печать, проверить вводимую команду (как такую, клавиша SELECT), или отказаться зафиксировать неверно нажатую клавишу. Возможна также некоторая "параллельная обработка" — если клавища не нажата, вы можете выполнить какое-либо другое задание. Наконец, эту функцию можно использовать для того, чтобы убрать курсор, или просто для ожидания нажатия клавиши. Смотри нежеследующие примеры — первый, простая задержка;

```
10 PRINT "Нажмите любую клавишу для продолжения"
20 IF INKEY$ = "  " THEN 20
```

Второй — стандартный ввод символа:

```
10 ch$ = INKEY$: IF ch$ = "  " THEN GOSUB 100:
   GOTO 10
20 ch$ = ch$ ' здесь начало обработки введенного кода
```

— здесь строка 100 содержит короткую программу, выполняемую, пока ожидается ввод символа.

См. также: INPUT, INPUT$ ( )

Цель: Читает порт ввода Z-80

Формат: *x* = INP (*номер порта*)

номер порта :: = номер порта Z-80
-32768 = < номер порта < = 65535

Действие: Выполняет команду ввода Z-80. Номер порта задается аргументом, который является целым числом без знака. Если необходимо запомнить номер порта в переменной целого типа, то вместо отрицательного значения используется двоичное дополнение, превышающее 32767: -32768 . . .–1 заменяются 32768 . . . 65535. Функция возвращает результаты ввода; так как Z-80 является 8-битным процессором, результаты, естественно, будут находиться между 0 и 255. Незадействованные номера портов чаще всего возвращают значение 255.

Для информации: номера портов ввода/вывода от (80) до (FF) резервируются для стандартных устройств ввода/вывода MSX. (См. техническое справочное руководство MSX). Номера портов от 00 до 7F зарезервированы для нестандартных аппаратных средств. Номера от (100) до (FFFF) не определены — это означает, что вся аппаратура, использующая слоты*) расширения, работает с картой памяти (и таким образом, слот можно выбирать) а не с портами ввода/вывода (что могло бы привести к конфликтам адресации).

*) слот — расширитель магистрали

Использование: Поскольку MSX-Бейсик непосредственно поддерживает работу с VDP, эта функция предназначена для нестандартных интерфейсов и для экспериментирования с определенными возможностями аппаратуры. Например, вы можете опросить состояние различных клавиш типа

"Shift", значение которых нельзя получить с клавиатуры. Вам может понадобиться осуществить нестандартный ввод через порт джойстика или переслать на него данные. Вы можете использовать индикаторы питания и фиксации верхнего регистра для выполнения некоторых специальных задач. В основном, этот оператор обеспечивает совместимость с другими версиями МБейсика.

Примечание: номера портов могут быть изменены в будущем, поэтому не используйте эту функцию в программах, предназначенных для тиражирования.

Пример:

```
10 A=INP (&HAA) AND &HF0 ' Защита битов порта, не связанных с клавиатурой
20 OUT &HAA, A OR 6: B=INP (&HA9) ' стробирование ряда клавиш №6 и ввод
30 IF B AND 1 THEN ' нажата клавиша Shift
```

См. также: OUT

Цель: Ввод из файла или с клавиатуры в режиме INPUT

Формат: INPUT [{ *номер файла* ,}] *список переменных* *"Подсказка"* ;

Номер файла :: = номер открытого файла, <= MAXFILES, 0 до 15
подсказка :: = любая строка символов, для вывода на экран
список переменных :: = одна или несколько простых переменных или элементов массива, разделенных запятыми

Действие: Этот оператор реализует основной способ ввода с клавиатуры или из файла. Если присутствует символ #, то ввод осуществляется из файла с заданным номером; при этом файл должен быть заранее открыт в режиме ввода INPUT. Если этот символ отсутствует, то оператор осуществляет ввод с калвиатуры в режиме ввода. Сначала рассмотрим действие режима ввода.

Вначале производится проверка на наличие подсказки, заключенной в кавычки и заканчивающейся точкой с запятой. Если она присутствует, то текст ее выводится на экран в том месте, где находится курсор; при этом происходит перемещение курсора. В любом случае на экран выдается вопросительный знак (?) и пробел; курсор передвигается при этом на две позиции. Курсор выводится на экран и режим ввода начинается.

Режим ввода продолжается до тех пор, пока не будет нажата клавиша RETURN. Вы можете набрать данные, сдвинуть курсор и использовать прямое редактирование. Когда бы вы ни нажали клавишу RETURN, вся строка, на которой находится курсор, прочитывается слева направо. Если курсор по-прежнему находится в той же

строке, в которой находился вопросительный знак, то все слева в этой строке — до вопросительного знака и пробела после него — автоматически исключается из строки; но если курсор находится на любой другой строке, учитывается вся длина. Помните: если вы вводите данные с клавиатуры и дойдете до конца строки, происходит переход на следующую строку, которая присоединяется к предыдущей и рассматривается как часть первой строки. Вся строка участвует в выполнении INPUT, и курсор помещается в начале следующей строки, что может послужить причиной сдвига экрана.

Если на данной строке были только пробелы, то INPUT предполагает, что вы не хотите изменять предыдущие значения, и они остаются прежними. Вообще этот оператор пытается преобразовать и присвоить первые же символы, найденные в строке, переменной из списка — при этом их типы должны соответствовать, иначе на следующей строке появляется сообщение об ошибке ?Redo from start (Повторите сначала), и в следующей строке вновь выводится подсказка и возобновляется режим ввода. Если используется числовой тип и вводимая величина слишком велика, то появляется сообщение Overflow (переполнение). Ошибка ?Redo не может быть перехвачена. Далее, у вас должно быть достаточно текста для обеспечения ввода значения каждой переменной из списка ввода. Если текста недостаточно, то Бейсик выдает подсказку ?? в следующей строке и переходит в режим ожидания, пока вы не введете дополнительных данных. Если текста слишком много, в следующей строке выдается сообщение ?Extra ignored (Лишнее игнорируется), что вызывает перевод строки.

Как это ни странно, если выдан запрос на ввод более одного элемента и во вводимом числовом

элементе присутствует двоеточие, Бейсик-система рассматривает это двоеточие как конец строки, в результате чего появляется подсказка ?? — запрос на продолжение ввода. Кроме того, вы можете заключить вводимые строки в кавычки; если первая кавычка является первым символом, присваемым строковой переменной, то переменной присваивается все содержимое кавычек; если кавычки — не первый встречающийся символ, то они являются частью символов строки. (Используйте кавычки для ввода ведущих и завершающих пробелов и запятых). Наконец, для числового ввода наиболее приемлемыми являются константы с символом & (&H, &O и &B) и математические обозначения (1.2e2, -5D + 03).

Для ввода из файла файл должен быть открыт в соответствующем режиме и иметь данные. Процесс чтения очень похож на ввод с клавиатуры, не считая того, что сообщения не выдаются. Данные, введенные из файла, должны быть в формате ASCII, каждый элемент введенных данных отделяется от другого пробелами (что приемлемо) или запятыми (лучше). Кроме того, каждая "строка" вводимых данных в файле должна ограничиваться признаком конца строки. Это означает, что приведенные ниже операторы PRINT и INPUT соответствуют друг другу:

```
100 PRINT #1, CHR$ (34); A$; CHR$ (34); ","; X;","; Y
200 INPUT #1, A$, X, Y
```

Чтобы создать выходной файл, в который можно будет ввести данные оператором INPUT, помните, что такой файл, если его распечатать, должен выглядеть так же, как если бы его ввели с клавиатуры в ответ на INPUT. Лишние кавычки и запятые увеличивают размер файла, но они также гарантируют, что какие бы данные

вы ни ввели туда оператором PRINT, они смогут быть прочитаны с помощью оператора INPUT.

Использование: INPUT при вводе с клавиатуры используется для совместимости и для простого, нерегулярного ввода простых элементов. Для опытного пользователя это быстро и просто. Для пользователя, которому нужна надежная программа, — это не приемлемо. Используйте LINE INPUT x$ — INPUT$ ( ) и x$ = INKEY$ — для более надежного ввода.

Для ввода/вывода файла этот режим используется чаще. Хотя данные в формате ASCII занимают большей объем памяти, программирование легче, за исключением оператора PRINT для вывода данных. Если данные загружены в память в соответствующем формате, INPUT считывает их быстро и эффективно.

Еще одно преимущество заключается в том, что INPUT может автоматически преобразовывать данные. Например, программа может определить функциональные клавиши &H, &O и &B так, что пользователь может задавать любое число в разных типах одним нажатием клавиши.

Следует обращать внимание на соответствие параметров и значений INPUT.

Пример:

```
10 INPUT "Гм . . .";  A, B: PRINT A, B
RUN
Гм .  . ? 123.4
?? авд
?Redo from start
Гм . . . ? e5, &B1101, асдор
?Extra ignored
0             13
Ok

10 ' файл уже открыт, пишет данные
20 Q$ = CHR$ (34): C$ = "," ' кавычки, запятая
30 ' пишет заголовок
40 PRINT #1, Q$; NAM$; Q$; C$; ZIP; C$; AGE; C$; Q$;
50 PRINT #1, COMMENTS$; Q$; C$; LIMIT
60 ' пишет файл документов
70 FOR X=1 TO LIMIT: INPUT #1, TRAN$ (X)
80 NEXT: CLOSE 1
. . .
200 ' читает открытый файл
210 INPUT #1, NAM$, ZIP, AGE, COMMENTS$, LIMIT
220 PRINT NAM$, ZIP, AGE, COMMENTS$, LIMIT
230 FOR X=1 TO LIMIT: INPUT #1, TRAN$ (x)
240 PRINT TRAN$ (X): NEXT: CLOSE 1
```

См. также: INPUT$ ( ), INKEY$, LINE INPUT, PRINT

Цель: Ввод за данного числа символов с клавиатуры или из файла.

Формат: *x$* = INPUT$ (*число* [, [ ] *номер файла*])

число :: = арифмеитческое выражение,
1 = < *число* < = 255
номер файла :: = номер открытого файла ,
= < MAXFILES от 1 до 15

Действие: Эта функция выполняет неформатный (как в фортране) ввод. Она похожа на функцию INKEY$, кроме того, что она выполняет многосимвольный ввод и работает с файлами данных. INPUT$ ( ) может быть использована двумя способами: С одним параметром она вводит с клавиатуры, с двумя параметрами она вводит из определенного уже открытого файла (в режиме INPUT).

В режиме ввода с клавиатуры курсор выводится на экран, причем он не может быть отключен. Затем накапливается заданное *число* введенных с клавиатуры символов. Когда ввод закончен (время не ограничено), то курсор отключается и функция INPUT$ ( ) возвращает строку символов. Ни один символ не выводится на экран, и только Control-STOP прерывает действие оператора. Точнее,

```
10 a$ = INKEY$ : IF a$ = "" THEN 10
```

и

```
10 a$ = INPUT$ (1)
```

идентичны, если не считать того, что последний выводит курсор.

Вариант, для ввода из файла, делает то же самое за исключением того, что ввод происходит без индикации действий: заданное число байтов считывается из открытого файла с заданным номером. Символы CR/LF (0D, 0A) в конце каждой строки вводимого файла воспринимаются как нормальные байты ввода.

INPUT$ ( ) воспринимает любой символ. Пауза, вызванная нажатием клавиши STOP, прерывается во время действия функции INPUT$ ( ) — вы не можете обнаружить условие STOP без использования специального приема (см. INP ( )), но условие Control-STOP всегда возвращает 3, если задействованы ON STOP/GOSUB/STOP ON. Кстати, указанный в ON STOP GOSUB номер строки не получит управления до тех пор, пока INPUT$ ( ) не введет последний символ и не передаст его по назначению, если какой-либо из этих символов равен Control-STOP (Прим.: Control-C также генерирует 3 без какой-либо передачи управления).

Использование: INPUT$ ( ) — один из наиболее аккуратных способов получения входных символов. Единственное его ограничение в том, что курсор выводится на экран, но он может быть удален тщательным подбором цвета во всех режимах, кроме SCREEN 0. Этот оператор обладает весьма высоким приоритетом в Бейсике, — здесь невозможно потерять символы. Иногда неудобно использовать его для многосимвольного ввода с клавиатуры, поскольку вы теряете контроль и не видите введенных символов до тех пор, пока не введен последний из них; вы можете, однако, использовать INPUT$ ( ) для ввода по одному символу (аналогично INKEY$).

Этот оператор удобен для ввода двоичных полей фиксированной длины из файлов данных.
Вы должны помнить, что все файлы данных, созданные с помощью PRINT, имеют в конце каждой физической строки символы CR/LF (возврат каретки/перевод строки —прим. нер.) аналогично файлам программ, записываемых на диск в формате ASCII. Следовательно, вы должны позаботиться об этом при создании файлов:

```
FOR x = 1 TO 1000: PRINT #1, LEFT$ (dat$ (x), 10) ;: NEXT X
```

создаст 10000 байтов данных при условии, что каждый dat$ ( ) по меньшей мере 10 байтов длиной, в виде 1000 10-байтовых полей. Поскольку в операторе PRINT в конце стоит точка с запятой (и нет числового вывода или разделяющей запятой), созданный файл представляет собой чистые данные, пригодные для ввода.

```
FOR x = 1 TO 1000: dat$ (x) = INPUT$ (10, 1): NEXT
```

Данные могут быть чем угодно, кроме кода конца файла: Control-Z или конец файла (1A), или 26 в досятичной системе счисления. Этот байт используется как признак конца файла для последовательных файлов и даже для INPUT$ ( ) (кроме функции прямого доступа в дисковом Бейсике). Во избежание потери части файла никогда не выводите чего-либо содержащего байт (1A) (кроме как с помощью BSAVE). Чтобы избежать этого, необходимо проверять байты данных во время создания файла. Замените все (1A) на последовательность ESC-NUL, т.е. на последовательность из двух байтов (1B) и (00). Так же замените каждый байт (1B) на последовательность из двух байтов (1B), (1B). Затем на вводе проверяйте последовательно каждый байт: если встретите ESC (1B), то отбрасывайте его и проверяйте следующий байт. Если это NUL (00) переведите его в Control-Z (1A), в остальных случаях пропускайте его без изменения. Это единственный способ обработки данной ситуации.

Пример:

```
100 VPOKE &H201F, &H4F ' изменить цвет курсора на экране на прозрачный
110 ON STOP  GOSUB 120: STOP ON: GOTO 130 ' нажаты клавиши Control-A
    и STOP
120 RETURN 140 ' не перехватывает Control-STOP,  он обрабатывается, как
    все клавиши
130 CH$ = INPUT$ (1) ' ввести следующий символ
140 CH = ASC (CH$)
150 GOSUB 2000 ' проверка введенного символа возвращает с = 1, если
    символ допустммый;  0 — если символ недопустимый;  2, если команда
160 IF NOT C THEN BEEP: GOTO 130 ' неправильно
170 IF C = 1 THEN PRINT CH$;: GOTO 130 ' допустимый дисплейный символ
180 ON CH GOTO ' здесь осуществляется обработка введенного символа
```

См. также: INPUT, PRINT, INKEY$

Цель: Определяет позицию подстроки в данной строке.

Формат: *x* = INSTR ([*начало* ,] *строка*, *подстрока*)

начало :: = номер начального символа в строке, 1—255
строка :: = любое допустимое строковое выражение
подстрока :: = любое допустимое строковое выражение

Действие: Эта функция исследует строку слева направо, начиная с нервого символа в строке (или с заданного начального символа) на предмет вхождения подстроки. Соответствне должно быть полным, тогда функция возвращает позицию первого вхождения. Если соответствующая подстрока не найдена, то функция возвращает ноль. Одна из строк может быть пустой, однако пустая подстрока соответствует чему угодно, а пустая строка не соответствует ничему.

Сообщение "Illegal function call", "Неправильный вызов фунцкии" возникает, если параметр "начало" выходит за границы строки, а сообщение "Type mismatch", "Несоответствие типов", если одна из "строк" нестрокового типа.

Использование: Эта функция может сравнивать, анализировать, сопоставлять и выделять строки. Пример использования — выбор на соответствие из списка команд. Рассмотрим следующий пример:

```
10 PRINT "Draw, Print, Change, Quit:";: I$ = INPUT$ (1)
20 ON INSTR (2, "xDdPpCcQq", I$)/2 GOSUB 100, 200,
   300, 400
30 BEEP: GOTO 10 ' недопустимий ответ
```


или вы можете проверить весь словарь

```
10 INPUT "слово для проверки (все клавиши) "; WD$
20 FOR x = 1 TO TP
30 IF INSTR (DICT$ (x), WD$) THEN x = TP: NEXT:
   GOTO 100
40 NEXT: BEEP: PRINT "No Match": GOTO 10
100 ' соответствует!
```

Пример:

```
(делит строку текста на отдельные слова в массиве)
10 LINE INPUT LN$: LN$ = LN$ + " ": CC = 1: WNUM = 0
20 IF CC = LEN (LN$) THEN 100 ' сделано
30 IF INSTR (CC, LN$, " ") = 1 THEN CC = CC + 1: GOTO 20
40 ND = INSTR (CC, LN$, " ")
50 WD$ (WNUM) = MID$ (LN$, CC, ND-1)
60 WNUM = WNUM + 1: GOTO 20
```

См. также: Разделы о строковых операциях в главах 2 и 4.

INT

фукция

Цель: Возвращает максимальное целое число, меньшее аргумента

Формат: *x* = INT (*ариф. выр.*)

ариф. выр. :: = любое арифметическое выражение

Действие: Функция вычисляет аргумент и преобразует его значение в целое число. Она не изменяет тип результата; она возвращает число того же типа, только без знаков после десятичной точки. Это осуществляется округлением до ближайшего меньшего целого числа, в отличие от FIX ( ), которая просто отбрасывает дробную часть.

Если аргумент строковый, возникает сообщение об ошибке "Type mismatch", "Несоответствие типов".

Использование: Эта функция по сути своей такая же, что и функция FIX ( ), за исключением обработки отрицательных значений. В приведенном ниже примере значения x будут различными:

x = ABS (INT (z)) и x = ABS (INT (–z))

Однако, агрумент редко бывает отрицательным; для уяснения вышесказанного см. описание функции FIX ( ). Функция позволяет проверить делимость, сравнивая результаты на равенство; число с дробью, даже отрицательное, не будет соответствовать значению целого числа, так что проверка правномерна. См. ниже:

Пример:

```
IF J/4 = INT (J/4) THEN PRINT "делится на 4!"
```

См. также: FIX ( ), CINT ( )

оператор

INTERVAL ON / OFF / STOP

Цель: Разрешает, запрещает или задерживает прерывание по таймеру.

Формат:

INTERVAL {ON / OFF / STOP}

Действие: Этот оператор управляет в Бейсике отсчетом реального времени для прерывания в регулярные интервалы. Прежде всего надо выполнить оператор ON INTERVAL = *тики* GOSUB *номер строки*, чтобы проинформировать Бейсик-систему, как часто (после какого количества тиков, 50 в сек.) Бейсик-система должна прерывать работающую программу для выполнения специальной программы обработки прерывания.

В отличие от него оператор ON не начинает и не прерывает процесса. Однако, если интервал и номер строки не были заданы, этот оператор не работает. Когда вы выполните оператор INTERVAL ON, то в заданном интервале в заданной строке начинается обработка прерывания. Оператор INTERVAL OFF отключает обработку прерываний. Оператор INTERVAL STOP выполняет что-то среднее: он приостанавливает обработку прерываний до следующего оператора INTERVAL ON.

Использование: Если вы хотите провести обработку прерываний, то необходимо использовать INTERVAL ON в соотвтествующей точке. Для прекращения обработки прерывания используйте INTERVAL OFF. Если у вас есть очень короткая срочная программа, использующая те же переменные, что и программа обработки прерывания, и вы не закончили обработку прерывания, то используйте INTERVAL STOP. Когда ваша короткая программа закончится, используйте INTERVAL ON, и отложенное прерывание выполнится.

INTERVAL ON / OFF / STOP

оператор

Внимание: таким способом можно отложить только одно прерывание, остальные будут потеряны. Помните, что неявный INTERVAL STOP выполняется на входе в программу обработки прерываний и отменяется на выходе из нее. Вы можете избежать этого при помощи явных INTERVAL OFF и INTERVAL ON в начале и в конце программ, так что длинная программа не закончится "вызывая себя" в бесконечном цикле из-за того, что займет больше времени, чем выбранный вами интервал!

Может оказаться полезным использование команд INTERVAL в других подпрограммах, чтобы не произошло слишком многое одновременно. Это позволит вам контролировать количество "одновременных" событий в работающей программе.

Пример:

```
10 INTERVAL = 20: GOSUB 1000: INTERVAL ON
...
100 INTERVAL STOP : GOSUB 500: INTERVAL ON ' чувствительная по
    времени программа
...
1000 INTERVAL OFF ' подпрограмма обработки прерывания
...
1090 INTERVAL ON: RETURN
```

См. также: ON INTERVAL

команда

KEY

Цель: Выводит на экран значение функциональных клавиш

Формат: KEY { *номер клавиши, строк. выр.* } LIST

номер клавиши .: = номер функциональной клавиши, функция которой подлежит изменению
1 = < *номер клавиши* < = 10

строк. выр. :: = любое строковое выражение — используются только первые 16 байтов

Действие: Первая форма этой команды: KEY [номер клавиши, строк. выр.] — используется для изменения значения любой из 10 возможных клавиш. Допускается любой текст в кавычках, любой тип строкового выражения; вычисляется выражение и первые 16 символов строки результата помещаются в область функциональных клавиш FNKSTR (F87F), имеющую по 16 байтов на каждую клавишу. При выполненни этой команды (если индикация клавиш разрешена), экран будет немедленно обновлен. Вывести на экран все строку KEY невозможно: размер строки изменяется в соответствии с установкой WIDTH, чтобы обеспечить по меньшей мере один пробел между каждым из 5 ключей, выводимых на экран. Ввод возврата каретки (0D) и кавычки (22) вполне возможен и довольно полезен в *строк. выр.* Другая форма в большей степени является командой: KEY LIST. Она выводит на экран 10 функциональных ключевых строк по порядку, по одной на строку. Поскольку вы не можете видеть всю строку по каманде KEY n, это является единственным способом точно выяснить, чему они равны. Разумеется, символы RETURN и другие, не выводимые на экран коды, действуют обычным образом при распечатке.

Использование: Эта команда дает возможность управлять функциональными клавишами Бейсика. Функциональная клавиша подставляет свою строку символов вместо любого отдельного кода клавиши. Следовательно, для запрещения действия любой функциональной клавиши используйте KEY n, " ". Если вам нужна клавиша для создания не выводимого кода, например, SELECT, тогда удобен оператор!

```
KEY n, CHR$ (24) .
```

Затем вы можете использовать INKEY$ или INPUT$ ( ) для чтения этой клавиши, как любой другой. Помните, что невозможно определить, была ли нажата функциональная клавиша или была введена соответствующая функциональная строка. Для блокировки функциональной клавиши без изменения соответствующей ему ключевой строки следует выполнить ON KEY GOSUB *номер строки* и KEY (*номер ключа*) ON — эти операторы, совместно блокируя функциональную клавишу, предотвратят ввод символов, не обращать к специальной подпрограмме обработки прерывания. Если вы все-таки изменяете строку выводимого ключа, вы можете вызвать INIFNK (003E), чтобы вернуться к стандартной ситуации.

Существует много способов использования функциональных клавиш. Они могут служить программным средством, могут содержать часто повторяющиеся тексты; они могут быть стандартными ответами, как YES, NO, MAYBE; они могут быть присвоены персональным данным, как имя пользователя; они могут быть использованы для выполнения различных функциональных подпрограмм, если нажать клавишу.

Кроме того, возможны комбинации вышеуказанных способов. Вы можете воспользоваться для сокращения строки соглашением, что в конце ее кавычки не требуются. Например,

```
KEY 2, "CLEAR5: CLOAD" + CHR$ (34) + "x"
```

выдает текст для ключа 2:

```
CLEAR5: CLOAD "x
```

при условии, что "x" — допустимое имя файла. Можно выполнить подряд две команды:

```
KEY номер клавиши, "CLOAD" + CHR$ (13) + "RUN"
+ CHR$ (13)
```

которые выполнят 2 команды:

```
CLOAD
RUN
```

Как вы можете видеть, символ кавычки должен быть специально добавлен к строковому выражению, как и символ CR; символ CR вызывает ввод команды, так же, как нажатие клавиши RETURN:

Пример:

```
10 FOR X = 1 TO 10: KEY X, " ": NEXT
20 KEY 3, CHR$ (254): KEY 5, CHR$ (255) ' используются только эти
   две клавиши
...
210 KEY 1, "COL.A": KEY 2, "COL.B": PRINT "выберите колонку"
220 A$ = INPUT$ (1): IF A$ < CHR$ (254) THEN BEEP: GOTO 210
230 ' используйте клавиши здесь
240 KEY OFF ' клавиши не показаны на экране, но они задействованы
...
1000 DEF USR = &H003E: A = USR (A) ' сброс функциональных клавиш
```

См. также: KEY (n) ON/OFF/STOP, ON KEY GOSUB

KEY(n) ON / OFF / STOP

оператор

Цель: Разрешает/запрещает/откладывает обработку прерывания по функциональной клавише.

Формат:

KEY (*номер клавиши*) {ON / OFF / STOP}

номер клавиши :: = допустимый номер функциональной клавиши
1 = < *номер клавиши* < = 10

Действие: Этот оператор отличается от KEY ON наличием скобок. Он используется для управления обработкой прерываний по функциональным клавишам. Предполагается, что вы предварительно выполнили оператор ON KEY GOSUB; в противном случае оператор KEY (n) . . . не действует.

Если задан KEY (n) ON, то нажатие клавиши, соответствующей функциональному ключу с заданным номером, прервет выполнение программы и передаст управление строке с заданным номером. Если позднее выполнен KEY (n) OFF, то обработка прерывания отменяется. Однако, если выполняется KEY (n) STOP, то обработка прерывания откладывается (при нажатии соответствующей функциональной клавиши) , и при следующем выполнении KEY (n) ON программа обработки прерывания немедленно выполнится.

При выполнении INPUT$ ( ), INPUT или LINE INPUT режим ввода не прерывается; любой другой ввод или вывод, осуществляемый подпрограммой, будет отложен до окончания режима ввода. Другими словами, режим ввода неявно вызывает KEY (n) STOP для всех функциональных ключей, а по окончании ввода отменяет. По сути дела, когда вы входите в подпрограмму обработки прерывания, то происходит то же самое, если не считать того,

что вы используете явные KEY (n) ON и KEY (n) OFF для преодоления этого.

Использование: Этот оператор управляет ходом программы. Вы можете использовать некоторые (но не все) функциональные ключи для осуществления обработки прерываний, и вы можете даже отложить выполнение или распознавание подпрограммы обработки прерываний. Запомните, KEY ON начинает распознавание и обработку прерываний; KEY OFF прекращает и то, и другое; KEY STOP распознает, но откладывает обработку до следующего KEY ON. Обращение к INIFNK (003E) восстанавливает значения всех ключей.

Пример:

```
10 ON KEY GOSUB 200, 300, 400
20 KEY (1) ON: KEY (2) ON: KEY (3) ON
...
30 FOR X = 1 TO 3: KEY (X) STOP: NEXT
40 GOSUB 1000' не теряйте функциональные ключи, но эта программа
   — срочная
50 FOR X = 1 TO 3: KEY (X) ON; NEXT
...
100 KEY (2) OFF ' не принимает функциональный ключ 2 с клавиатуры,
    недопустимо
...
180 KEY (2) ON ' теперь допускается ввод функционального ключа 2
```

См. также: ON KEY GOSUB, KEY ON/OFF/LIST

KEY ON / OFF

оператор

Цель: Включает или выключает строку подсказки функционального ключа.

Формат: KEY {ON / OFF}

Действие: Этот оператор воздействует на последнюю строку экрана. KEY ON включает вывод функционального ключа на нижнюю строку экрана, если ключ был выключен; KEY OFF выключает, если он был включен. В других случаях оператор не действует. Если вывод на экран включен, то величина экрана 23 строки, а если нет — то 24 строки. Когда 24-ая строка возвращена выключением вывода на экран функционального ключа, она пустая. KEY OFF никогда не вызывает сдвиг экрана или перемещение курсора — если команда введена на 23-ей последней строке, функциональная строка будет удалена прежде, чем курсор перейдет на "следующую строку". KEY ON может сдвинуть экран, если курсор находится в 24 (23) строке.

Вывод на экран подсказки функционального ключа состоит из 5 строк текста, разделенных пробелом, означающим, что текст соответствует каждому из 5-ти функциональных ключей. (Естественно, команда WIDTH воздействует на длину этих текстов). Когда нажата клавиша SHIFT, на экран выводится вторая группа из 5-ти текстов, соответствующая функциональным ключам F6—F10, которые представляют собой измененные ключи F1—F5. Оператор KEY (n) влияет на содержимое этих подсказок.

Использование: Помните, что этот оператор не влияет на использование функциональных ключей. Разумеется, он позволяет увеличивать размер экрана. Он также удаляет с экрана отвлекающий текст клавиш, если вы уже запретили ключи оператором KEY (n) OFF, или если вы просто не ввели новые значения для ключей.

Следует также помнить, что вы не обязаны использовать функциональные ключи по прямому назначению — они используются для формирования строки подсказки, так что KEY ON/OFF включает и выключает подсказки, не вызывая сдвига экрана. Подсказки функциональных ключей не используются в графических режимах SCREEN 2 и 3.

Пример:

```
10 KEY OFF: GOSUB 1000 ' запрос, какую функцию исполнить, Возвращает ответ
20 ON ANS GOSUB 1010, 1011, 1012 ' Выполнение команд установки KEY n, " " cmds
30 KEY ON ' теперь вывод обновленных функциональных ключей
```

См. также: KEY (n) ON/OFF/STOP, ON KEY (n) GOSUB

LEFT$

фукция

Цель: Возвращает самые левые n символов строки.

Формат: *x$* = LEFT$ (*строк. выр.*, *длина*)

строк. выр. :: = любая строковая константа или выражение
длина :: = количество символов
0 < = *длина* < = 255

Действие: Эта функция возвращает строку, содержащую заданное число самых левых символов данной строки. Если "*длина*" больше длины "*строк. выр.*", возвращается вся строка. Если "*длина*" равна 0, возвращается пустая строка, и если "*строк, выр.*" является пустой строкой, то также возвращается пустая строка.

Если "*длина*" превышает допустимую или пропущена, то возникнет сообщение "Illegal function call", "Неправильный вызов функции": если перепутан порядок следования параметров, то возникнет сообщение "Type mismatch", "Несоответствие типов".

Использование: LEFT$ (a$, b) идентична MID$ (a$, 1, b) однако, она экономит по меньшей мере 2 байта или один нажим клавиши. Кроме того, эта функция более понятна. Функция LEFT$ ( ) симметрична функции RIGHT$ ( ).

Пример:

```
PRINT LEFT$ ("abc", 2); LEFT$ (LEFT$ ("def", 2), 1)
abd
Ok
```

См. также: MID$ ( ), RIGHT$ ( ), раздел главы 2 о строковых операциях.

LEN

Цель: Возвращает длину строки символов.

Формат: *x* = LEN (*строк. выр.*)

строк. выр. :: = любое строковое выражение

Действие: Функция возвращает длину заданной строки в байтах. Это означает, что LEN ( ) может вернуть от 0 (для пустой строки) до 255 (для максимально длинной строки).

Использование: Во-первых, эта функция позволяет вам определить, не является ли строка пустой до выполнения функции, которая не может выполняться на пустой строке. Затем она позволяет работать со строками различной длины, рассматривая их как заполненные, и всегда ссылаясь на их длину. Наконец, некоторые строковые манипуляции легче выполнять, если вы знаете, какой длины строка. См. следующие примеры:

Пример:

```
100 IF LEN (A$) > 0 THEN TYPE$ = MID$ (A$, 1, 1)

100 IF LEN (S1$) < LEN (S2$) THEN SWAP S1$, S2$
110 S2$ = STRING$ (LEN (S1$) – LEN (S2$), " ") + S2$

100 IF LEN (CARD$) = 80 THEN GOSUB 1000 ' проанализировать карту
    ввода
```

См. также: MID$ ( ), LEFT$ ( ), RIGHT$ ( ), STRINGS ( ), Глава 2 о строках

Операции

Цель: Основной оператор присваивания

Формат: [LET] *переменная* = *выражение*

Переменная :: = любая новая или существующая переменная, элемент массива или MID$ ( ) = функция.
выражение :: = любое выражение; должно быть либо строковым, либо арифметическим того типа, что и "*переменная*".

Действие: Это стандартный способ выполнения вычислений в Бейсике. Выражение вычисляется, его тип преобразуется, если возможно, и результат присваивается переменной. Попытка присвоить строковое выражение арифметической переменной, или наоборот, вызовет сообщение об ошибке "Type mismatch", "Несоответствие типов". Здесь могут произойти любые ошибки вычисления выражений.

Использование: Любой из существующих вариантов Бейсика очень редко требует использования слова LET. Предполагается, что если оператор состоит из неключевого слова, за которым следует знак равенства (=), то это оператор присваивания (LET).

Следует заметить, что значение выражения справа от знака равенства присваивается переменной слева от знака равенства, т.е. знак равенства в данном случае означает только операцию присваивания. Единственным исключением является случай, когда справа от знака равенства стоит текстовая константа. Эти операции просто копируют в переменную длину строки и указатель данных из выражения. Если соответствующие строковые данные изменены, например, посредством POKE, то все переменные, указывающие на данные, дадут измененные результаты. Этого можно избежать:


```
a$ = b$ + " " или  a$ = "константа" + " "
```

Если вы напишете

```
FOR x = 1 TO 1000: ae$ (x) = ".....": NEXT
```

для инициализации строкового массива, то затем обнаружите, что для хранения инициализированных переменных не было выделено намяти; это произошло потому, что каждый элемент занимает 3 байта: длина + указатель — все указывают на одну и ту же строковую константу в теле программы! Добавьте + " ", чтобы справиться с этой проблемой (разумеется, за счет времени и строкового пространства).

Пример:

```
10 LET X = 1: LET Y = 2: Z = 3
20 A$ = B$+ " " : D$ = E$ + G$
```

См. также: Глава 2 выражениях и переменных.

Цель: Рисует линии и прямоугольники на графическом экране.

Формат:

LINE [[@][STEP] (*x1*, *y1*)] — [@] [STEP]
(*x2*, *y2*) [[, *цвет*] , { B }]
BF

x1, *x2* :: = любое допустимое значение координаты X (номер столбца)

y1, *y2* :: = любое допустимое значение координаты Y (номер строки)
все точки находятся в диапазоне: -32768 =< *точка* =< 32767, где (0,0) верхний левый угол экрана, причем *x* растет вправо, а *y* вниз

Цвет :: = любой допустимый номер цвета 0 =< *цвет* =< 15: если пропущен, то используется установленный прежде цвет изображения. См. COLOR

B :: = рисует прямоугольник с заданными верхней левой; нижней правой вершинами.

BF :: = рисует прямоугольник и полностью его закрашивает.

STEP :: = координаты следующей точки вычисляются относительно координат предыдущей точки.

@ :: = только для документации: не имеет никакого эффекта.

Примечание: Если пропущены координаты первой точки, то берутся координаты точки, использованные в последнем выполненном графическом операторе. Если пропущен параметр "цвет" и использованы параметры B или BF, то запятая перед ними обязательна.

Действие: Этот оператор является наиболее распространенным способом рисования на графическом экране. Если не установлен режим SCREEN 2 или 3, то обращение к оператору LINE вызовет сообщение об ошибке "Illegal function call", "Недопустимый вызов функции". Если же режим SCREEN 2 или 3 установлен, то этот оператор рисует линии, вычерчивает прямоугольник или закрашенный прямоугольник в зависимости от наличия (или отсутствия) параметров B или BF.

Если указаны 2 точки, они относятся к началу и концу изображаемого объекта (верхняя левая) нижняя правая точка (не забудьте указать тире!), то используется последняя графическая точка, изображенная ранее в вашей программе (начальной считается точка (0, 0) при включении ЭВМ; координаты точки не сбрасываются командой RUN). Более того, эта точка может быть абсолютная или относительная (последнее — если указан параметр STEP) — если относительная, то заданные координаты прибавляются к координатам предыдущей графической точки (где бы она ни была упомянута), и результат сложения есть координата точки для рисования от/до. Символ @ используется только для совместимости с другими версиями Бейсик и не имеет абсолютно никакого эффекта.

Если "цвет" не указан, то рисунок имеет цвет изображения, заданный в COLOR; в противном случае может быть указан любой цвет. Если не заданы параметры B и BF, то рисуются прямые линии; при параметре B рисуется прямоугольник; при BF рисуется заполненный прямоугольник — самый быстрый способ нарисовать фигуру в MSX-Бейсике.

Все точки существуют в системе координат от -32768 до 32767 в обоих направленях; на

экране отображаются только координаты X от 0 до 255 и Y от 0 до 191. Любая часть рисунка за пределами этого окна будет отображена без потери скорости, часть рисунка, расположенная на экране, сохраняется. Это касается и варианта BF.

Использование: Этот оператор обеспечивает большую подвижность при создании рисунков на экране. Три главные назначения — рисование прямых линий, прямоугольников и заполненных прямоугольников, не все которых обязательно появятся на экране. При пропущенной первой точке некоторые линии или другие фигуры могут быть связаны между собой. Более того, относительный вариант STEP допускает вычерчивание фигур без ссылки на абсолютные координаты — например, чтобы позволить подпрограммам помещать один и тот же объект в различные места. Кроме того, правильное использование фактора размера позволит задавать объект в масштабе.

```
1000 LINE-STEP (SCALE*3, SCALE*4),, BF: RETURN
```

— эта последовательность нарисует заполненный прямоугольник в текущем цвете курсора (изображения) и от текущей позиции графического курсора вниз и вправо с шириною 3/4 высоты и размером, заданным масштабом.

```
1000 RESTORE 1020: READ ND: FOR Z = 1 TO ND:
     READ X, Y
1010 LINE-STEP (X, Y): NEXT: RETURN
1020 DATA 3, 10, 0, -5, -5, -5, 5
```

рисует треугольник вправо/вверх от графического курсора. Кроме того, для управления размером фигуры можно умножить X и Y на масштабный множитель. В вышеуказанных случаях для установки графического курсора в "начало" объекта (если его там еще нет) используйте PSET.

Параметр BF является самым быстрым методом в MSX-Бейсике для заполнения фигуры. Несколько операторов LINE ... BF эквивалентны по времени одному оператору PRINT.

Пример:

```
100 ' рисует пятиугольник, разделенный на сегменты, размер задан
    масштабом . . .
110 ' начало задано xb, yb и цвет cl
120 RESTORE 170: SCREEN 2: COLOR 9
130 FOR X = 1 TO 5: READ A, B
140 LINE (XB, YB) – STEP (A*SC, B*SC), CL: NEXT
150 RESTORE 170: FOR X = 1 TO 5: READ A, B
160 LINE – (A*SC + XB, B*SC + XB), CL: NEXT: RETURN
170 DATA 10, 2, 6, 10, -6, 10, -10, -2, 0, -10
```

См. также: PSET, PRESET, POINT ( ), DRAW, CIRCLE, Главу 4 о графике.

Цель: Вводит текст из файла или с клавиатуры до нажатия клавиши RETURN

Формат: LINE INPUT [{#*номер файла*}] *строк пер.*
"строк. конст.";

строк. пер. :: = любая строковая переменная или элемент строкового массива
номер фойла :: = номер окрытого файла 0—15
строк. конст. :: = строковая константа

Действие: Этот оператор имеет два варианта. Если задан номер файла, то читаются символы из открытого файла с заданным номером. В противном случае читаются символы с клавиатуры. В любом случае оператор может прочесть до 255 символов, помещая их в "*строк. пер.*" Процесс ввода может остановить только ввод символа RETURN (0D) или считывание из файла признака конца файла. (1A). Любой другой символ рассматривается как часть строки ввода.

При вводе с клавиатуры (без символа #) выводится на экран подсказка "константа", и курсор сдвигается на следующую за ней позицию. При этом устанавливается режим ввода, и вы можете вводить любую комбинацию символов, (используя любые способы редактирования с клавиатуры). При нажатии клавиши RETURN первые 255 символов, начиная с начальной позиции курсора и до ввода конца строки (несколько физических строк), вводятся в строковое пространство, и этот текст станет новым значением "*строк. пер.*".

В операторе LINE INPUT нет ничего, что могло бы помешать ввести слишком много текста, и нет ничего, что удершивает курсор в одной строке, т.к. возможен сдвиг экрана. Этот оператор принимает все без выдачи сообщения об ошибке. Файловая версия оператора (с символом #) читает последовательно символы, пока не будут прочитаны 255 символов, либо RETURN (0D), либо не будет встречен конец файла (1A). Эти заключительные символы опускаются; Control — Z (1A) означает физический конец файла.

Нажатие Control—C или Control—STOP вызовет прекращение действия оператора LINE INPUT, перевод интерпретатора в прямой режим и выдачу сообщения "Break" (Прерывание). Если вы используете режим ввода из файла, действует только Control—STOP: при этом выдается сообщение "Device 1/0 error", "Ошибка устройства ввода/вывода".

Использование: Этот оператор один из наиболее удобных операторов ввода/вывода — он дает удачное сочетание легкости использования и защиты от неожиданных ошибок. Возможности управления курсором в этом операторе отсутствуют. Если вы вводите число, то для его преобразования после чтения используйте функцию VAL ( ). Процедура входного контроля данных задается по мере необходимасти.

Файловая версия позволяет поместить всю запись в строковую переменную. Разумеется, символы "конец файла" Control—Z (1A) или код возврата каретки RETURN (0D) вызывают конец ввода; кроме того, ввод прерывается после введения 255 символов. Лучшее использование LINE INPUT — ввод "записи" — строки текста или программы (в формате ASCII), которая была записана ранее.
Помните, что символ RETURN и все следующие за символом перевода строки (0A) будут полностью потеряны. Сторока подсказки в варианте ввода с клавиатуры выводится на экран; ввод начинается с позиции после последнего символа подсказки. Символы на экране при выводе подсказки оператора LINE INPUT стираются ею.

LINE INPUT

оператор

Пример:

```
100 LINE INPUT "Напечатать имя программы:"; PN$
110 OPEN "CAS:" + PN$ FOR INPUT AS #1
120 IF EOF (1) THEN PRINT "До свиданья!": END
130 LINE INPUT #1, LN$
140 LPRINT LN$: GOTO 120
```

См. также: INPUT, INKEY$, EOF ( )

команда

LIST

Цель: Вывод текста программы на экран или на печатающее устройство (принтер)

Формат: [L] LIST [*ном. строки*] [– *ном. строки*]

ном. строки :: = любой номер строки от 0 до 65529 — может быть несуществующим или точкой (.)

L :: = вывод на принтер, а не на экран

Примечание: Точка (.) обозначает "текущий" номер строки.

Действие: Эта команда, которая может быть оператором программы, выводит текст программы, находящейся в настоящий момент в памяит. Если указан LIST, то вывод происходит на экран: если используется LLIST, то текст отсылается на печатающее устройство. Если не указаны параметры, то выводится вся программа. (При выводе программы кодированный текст преобразуется в удобную для человека форму, при этом могут быть незначительные различия с тем, что было введено) . Листинг распечатывается таким образом, что после номера строки ставится один пробел, но не выполняется никаких других преобразований, чтобы сделать текст "изящнее" — текст распечатывается в форме настолько близкой к той, которая есть в действительности, насколько это возможно.

Если задан единственный номер строки, то распечатывается только зта строка. Тире до или после этого номера означает, что будет распечатано все до или после этого номера строки. Два номера строки, разделенные тире, указывают, что должны быть распечатаны только строки с номерами, находящимися внутри данного дпапазона. Чтобы представить "текущий" номер строки, вы можете использовать точку, — имеется в виду последний, использованный в команде или введенный номер.

При использовании в качестве оператора в выполняемой программе LIST переводит MSX-Бейсик в прямой режим. В команде LIST могут использоваться несуществующие номера строк, при этом не будет ошибки. Для паузы печати может использоваться STOP, а Control-STOP прекратит выполнение программы.

Использование: Этот оператор используется тогда, когда вы хотите посмотреть на то, в каком виде ваша программа в действительности хранится в памяти. Он выводит текст программы в соответствии с номерам и строк и может вывести текст на печатающее устройство. LLIST не проверяет наличие устройства; однако, вы можете прервать LLIST с помощью Control-C и получить ошибку устройства ввода-вывода.

Пример:

```
200 REM  Это последняя
150 REM  Это средняя
100 REM  Это первая
LIST .
100 REM  Это первая
Ok
LIST -175
100 REM  Это первая
150 REM  Это средняя
Ok
```

См. также: DELETE, RUN, NEW

Цель: Ввод Бейсик-программы, хранящейся в формате ASCII

Формат: LOAD *имя устр. с файлом* [, R]

имя устр. с файлом :: = допустимое строковое выражение, указывающее имя существующего устройства ввода и имя файла, который следует загрузить.

R :: = указывает на то, что после загрузки следует выполнить программу

Действие: Эта команда может использоваться для загрузки Бейсик-программы, предварительно записанной командой SAVE или записанной в виде последовательного файла данных. Она может также использоваться для загрузки цепочки программ — операнд R прогоняет загруженную программу, как только она будет полностью введена и держит все файлы открытыми (в отличие от RUN *устройство — имя файла*, который в других отношениях идентичен). MSX-дисковый Бейсик использует этот оператор для загрузки всех программ — а не только имеющих формат ASCII — однако, стандартный MSX-Бейсике единственным допустимым устройством является CAS:, которое предполагается, если пропущен тип устройства и указано имя файла.

Если дано только имя устройства и нет ни одного имени файла, то предполагается, что последовательный текстовой файл, обнаруженный на устройстве, является единственным, который следует загрузить. Если не используется ни имя устройства, ни имя файла, возникает ошибка "Missing operand", "пропущен операнд"; неисправное устройство или пустое либо недопустимое имя файла без устройства вызывает сообщение "Bad file name", "Неправильное имя файла".

LOAD

команда

Использование: Во-первых, эта команда используется как команда загрузки программы, хранящейся в текстовом формате ("ASCII") на ленте (или на дополнительном устройстве, не существующем в стандартной системе). Этот формат представляет собой строго последовательный текст, написанный в символах ASCII, и каждая пронумерованная строка ограничивается парой RETURN/Line Feed (перевод каретки), записанной с помощью SAVE или PRINT в последовательный файл вывода. Таким образом, у вас есть средство, которое размещает программу, сохраняя результаты на ленте; вы можете затем использовать LOAD и прочитать конечный результат (а, возможно, использовать CSAVE, чтобы сохранить пространство и время). Из выполняемой программы цепочка може быть построена только к программам, хранящимся в формате ASCII. RUN devfilename очищает все, тогда как LOAD имя устр. с файлом сохраняет любые открытые файлы ввода/вывода.

Помните, что дисковый Бейсик использует этот оператор для загрузки любых программ. Стандартный MSX-Бейсик ограничивается вводом программного файла ASCII с кассеты.

Пример:

```
. . .
15000 LOAD "CAS: MOD2", R ' загрузка следующего программного
      модуля, который следует выполнить

LOAD "CAS:" ' загрузка следующей программы, записанной на ASCII,
с ленты.
```

См. также: SAVE, CLOAD, CSAVE, BLOAD, BSAVE, RUN

оператор

LOCATE

Цель: Установка курсора в любом месте текстового экрана, его исчезновение

Формат: LOCATE [*x*], [*y*], [*ключ*]

x ::= x-позиция (столбец), в который следует поместить курсор
y ::= y-позиция (строка), в которую следует поместить курсор
ключ ::= числовое выражение: 0 = курсора нет, иначе — есть

Примечание: Все значения описанных выше переменных должны находиться в пределах 0<=*x*<=255, в противном случае возникает "Illegal function call", "неправильный вызов функции". Кроме того, любой элемент может быть пропущен, но его запятая должна присутствовать перед любыми следующими элементами.

Действие: Этот оператор управляет курсором в текстовых режимах — SCREEN 0 и 1. Он воспринимается и в других режимах, но его действие сохраняется и используется в следующем появляющемся текстовом экране.

Строки текстового экрана нумеруются от нуля (верх) до 22 (с отображением функциональных ключей) или 23 (без отображения функциональных ключей) вниз. Аналогично, столбцы нумеруются от нуля (слева направо) до последней колонки — этот размер указывается в операторе WIDTH. LOCATE перемещает курсор в указанное знакоместо. Если *x* или *y* пропущены, предыдущее значение сохраняется. Если "*ключ*" и он равен нулю, то он выключает курсор, который MSX-Бейсик всегда стремится отобразить. Любое другое значение (например, 7) снова вернет курсор. Помните, что любой тип оператора INPUT заново включит курсор

во время выполнения оператора. Однако, INKEY$ позволяет курсору оставаться выключенным; это единственный способ гарантировать, что курсор никогда не появится, а также не появятся входные данные с клавиатуры. Следует заметить, что MSX-Бейсик всегда отображает и обновляет курсор при печати (PRINT) или вычислениях; LOCATE позволяет удалить неприятное мерцание.

Использование: Курсор, который изменяется данным оператором, хранится в отдельной ячейке от графического курсора, и, таким образом они не влияют один на другой с помощью этого оператора. Функции CSRLIN и POS ( ) используют коорбинатся, устамовленные LOCATE. Если используются строка или столбец с номерами больше, чем максимально отображаемые, но меньше 255, то ошибки не происходит, но курсор передвигается на максимально возможную позицию в этом направлении без какого-либо возврата.

Пример:

```
100 LOCATE 0,22: LINE INPUT "Chose 1, 2 or 3:"; CH$
110 LOCATE 0,0,0 ' во избежание любого сдвига экрана
```

См. также: CSRLIN, POS ( ), WIDTH, SCREEN

Цель: Вычисление натурального логарифма

Формат: *x* = LOG (*числ. арг.*)

числ. арг. :: = любое числовое выражение: должно быть положительным, больше нуля.

Действие: Эта функция оценивает свой аргумент и выдает ошибку "Illegal function call", "неправильный вызов функции", если он равен нулю или отрицательный. В противном случае она вычислит натуральный логарифм (по основанию e) этого аргумента с двойной точностью.

Использование: LOG ( ) используется для любого логарифма. Например, логарифм по основанию 10 вычисляется следующим образом:

```
LOG (arg) / LOG (10)
```

Следует заметить что e = 2,7182818284590, до 14 разряда. Однако, поскольку при такой записи получается, что MSX-Бейсик использует для младших разрядов 88, а не 90, ошибка составляет 2e—13. Эта ошибка приводит к тому, что младший (14-ый) разряд в операциях EXP ( ) , LOG ( ) и возведения в степень будет неточным; для абсолютной точности ограничьтесь 12-м и 13-м разрядами, в противном случае игнорируйте это.

Пример:

```
?LOG (EXP (1))
.99999999999986 ← заметьте, что ошибка мала; должно быть 1
Ok
```

См. также: EXP ( )

LPOS

фукция

Цель: Указывает позицию печатающей головки MSX-принтера.

Формат: *x* = LPOS (1)

Примечание: в качестве параметра этой функции может использоваться что угодно, но что-нибудь должно быть задано обязательно!

Действие: MSX-Бейсик ведет от счет позиции нечатающей головки подключенного принтера MSX. Любой напечатанный с помощью LPRINT символ обновляет его. Следует заметить, однако, что Бейсик не знает о некоторых специальных функциях принтера, которые могут повлиять на позицию печатающей головки; поэтому эта функция надежна только для прямой печати текста.

Использование: LPOS ( ) оказывает большую помощь при печати столбцов данных на строчном принтере. Вы также можете проверить, в каком положении остался принтер после выполнения подпрограммы; при прохождении данного столбца для точного размещения принтеру можно послать RETURN.

Пример:

```
250 IF LPOS (1) > 50 THEN LPRINT
260 LPRINT TAB (60)
```

См. также: POS ( ), PRINT, LPRINT

оператор

MAXFILES =

Цель: Резервирование определенного числа буферов управления файлом

Формат: MAXFILES = *арифметическое выражение*

арифметическое выражение :: = любое арифметическое выражение: 0 = < арифметическое выражение < = 15

Действие: Эта псевдопеременная, предназначенная только для записи информации, используется для установления максимального числа одновременно открытых файлов — и, соответственно, числа зарезервированных блоков управления файлом (БУФ) и буферов данных. Каждый БУФ трерует дополнительные 267 байт памяти.

Выполнение MAXFILES = включает подразумеваемый оператор CLEAR. Если параметр не удовлетворяет ограничениям, выводится сообщение "Недозволенное обращение к функции"; если недостаточно памяти (в памяти еще и программа), происходит ошибка "выход за пределы памяти". Если ничего такого не случится, память резервируется, как требуется: только очистка при включении питания или другой оператор MAXFILES = перераспределят зту память.

Использование: MAXFILES = 1 устанавливается по умолчанию при включении питания. БУФ всегда доступен; MSX Бейсик может использовать его для внутренних целей. Тем не менее, каждый номер файла, используемый в операторе OPEN, должен иметь буфер и БУФ, зарезервированные для него; MAXFILES = выполняет это.

Проверив ячейку MAXFILE (F85F), вы можете определить текущую установку MAXFILES = . Но помните — MAXFILES всегда очищает все ваши переменные.

Пример:

```
10 MAXFILES = 2 ' 1-источник, 2-приемник
20 CLEAR 550, &HF000 ' данный оператор резервирует пространство
   для символьных строк, не оказывает влияния на результат работы
   MAXFILES
```

См. также: CLEAR, OPEN, приложение "Карта памяти".

Цель : Объединяет текст программы с ленты или диска с программой в памяти

Формат: MERGE "имя устройства [имя файла]"

имя устройства : : = CAS: ....
[имя файла] : : = строковое выражение, содержащее 6 символов, определяющее имя файла. Если имя файла пропущено, то загружается первый файл.

Действие: Эта команда загружает программу с указанного устройства и объединяет ее с программой в памяти. Загружаемая программа должна храниться в формате ASCII (см. LOAD, SAVE). Любые нумерованные строки текста будут заменены соответствующими строками загружаемой программы; любые несовпадающие строки сохраняются. MERGE не быстрое, но эффективное средство.

Смотрите LOAD об ограничениях на имя устройства и имя файла. Эта команда может использоваться как оператор в вашей программе, но ее выполнение переводит компьютер в режим непосредственного выполнения.

Использование: Эта команда позволяет вам выбирать фрагмент из одной программы — скажем, используя DELETE и RENUM, затем SAVE (сохранить) его, так что вы можете перезагрузить его и объединить с другой программой. Этот оператор выполняет CLEAR а во время своей работы, так что его использование в программах несколько осложняется. Вы можете лучше использовать его для помощи при создании библиотеки подпрограмм на ленте — затем только MERGE (объединить) желаемые подпрограммы по именам с вашей головной программой.

Пример:

```
CLOAD "prog 1"
1   REM главная
2   главная в prog 1
99  REM
109 REM подпрограмма
110 подпрограмма, которую вы желаете для prog 2
9999 REM
DELETE 1–99 (уничтожить главную в prog 1)
SAVE "CAS : In100",A' (заметьте, что файл сохраняется в ASCII
формате)
CLOAD "prog 2"
MERGE "CAS : In 100"
CSAVE "endprg"
```

См. также: SAVE, LOAD

Цель: Извлечь или модифицировать часть символьной строки

Формат: *x$* = MID$ (*стрвыр*, *начсимв* [, *длина*])
или
MID$ (*стрпер*, *начсимв* [, *длина*]) = *стрвыр*

стрвыр :: = любое строковое выражение
начсимв :: = любое числовое выражение, дающее 1—255, задающее код начального символа
длина :: = любое численное выражение, из диапазона 1—255, определяющее число символов в строке
стрпер :: = любая допустимая строковая переменная или элемент массива

Действие: Эта функция, может использоваться в обеих частях оператора присваивания и обепечивает легкий доступ к любой символьной строке. Первый вариант извлекает из строкового выражения столько символов, сколько задано длиной, начиная с *начсимв*. В области памяти строк создается новая строка, которая возвращается, как результат функции. Конечно, не может вернуться больше символов, чем осталось от *начсимв* до конца строки. Пустая строка всегда выдает пустую строку. Если длина не указана, подразумевается остаток строки (независимо от его длины).

Второй вариант, MID$ слева от оператора присваивания, очень важен, т.к. позволяет проводить строковые операции на том же месте — без использования дополнительной памяти: существующие строковые данные модифицируются в том же месте, где они хранились. В этом случае, если длина указана, результат строкового выражения справа от знака присваивания должен быть точно той же длины. Если длина пропущена, она предполагается равной

длине строкового выражения, но ни в коем случае не больше, чем длина остатка уже существующей строки. Этот вариант вычисляет строковое выражение и замещает им существующие символы в строковой переменной, исходя из *начсимв* и *длины*. Если длина не указана, могут быть переписаны любое число символов (или ни одного).

Использование: Эта функция исключительно полезна в разделении строк и объединении их затем вместе. Следует иметь в виду требование дополнительного пространства, существенное для = MID$ варианта этой функции. MID$ = можно использовать для всех строковых присваиваний, чтобы избежать "сборки мусора". Вместо a$ = *стрвыр*, используйте MID$ (a$, 1) = *стрвыр*, чтобы избежать использования дополнительной памяти. Будьте аккуратны при определении размеров строковой переменной — переменная уже должна существовать и быть достаточно длинной, чтобы все поместилось.

Пример:

```
100 A$ = STRING$ (20, "  ") : MID$ (A$, 1) = Z$+X$+" : "
110 PRINT MID$ (A$, 10, 5)   ' использует дополнительно 5 байт памяти
```

См. также: LEFT$ ( ), RIGHT$ ( ), STRINGS ( )

Цель: Включить или выключить мотор кассетного магнитофона

Формат: MOTOR { ON / OFF }

Действие: Этот оператор включает или выключает управляющее реле перемотки кассет. Только очистка при включении питания и функции кассетного B/Bb могут изменить ето.

Использование: Этот оператор включает или отключает реле, контакты которого могут переключать маленький ток — около 150 mA — достаточный для подачи питания на мотор кассетного магнитофана. Поскольку требуемые контакты доступны в разъеме на любом MSX — компьютере, очевидно и другое применение. Полезно разрешить пользователю перематывать ленту к нужной позиции, что невозможно в большинстве стандартных магнитофонов при выключенном моторе.

См. также: Описание подключения к магнитофону в руководстве пользователя.

NEW

команда

Цель: Уничтожить программу в памяти

Формат: NEW

Действие: Эта команда уничтожает всю программу в памяти, все переменные и закрывает все открытые файлы.

Использование: Используйте эту команду для уничтожения программ, прежде чем начать вводить новую программу. Иначе только "измените" вашу старую программу.

Пример:

```
NEW
```

См. также: CLEAR, DELETE, LIST, RUN

оператор

NEXT

Цель: Указать конец цикла

Формат: NEXT [*последовательность переменных*]

последовательность переменных :: = одно или несколько имен переменных, разделенных запятыми

Действие: Этот оператор указывает конец цикла, начатого предшествующим FOR оператором. Если нет имени переменной, завершается самый внутренний цикл. Если вы перечислите имена переменных цикла, соответствующие циклы завершатся в этом месте.

Использование: NEXT всегда используется с FOR

Пример:

```
10 FOR I=0 TO 1
20 FOR T=0 TO 2
30 FOR K=0 TO 3
40 NEXT K, J, I
```

См. также: FOR

OCT$

фукция

Цель: Преобразует целое в восьмеричный формат

Формат: *x$* = OCT$ (*целвыр*)

целвыр :: = любое арифметическое выражение, имеющее результатом целое число

Действие: Эта функция вычисляет аргумент, преобразует результат в целое, знаковое или беззнаковое; если не в диапазоне — 32768—65535, происходит ошибка "Переполнение". Иначе результат преобразуется в символьную строку, представляющую его значение в восьмеричной системе. Пробелы перед числом и после него не создаются, более того, пробелы перед числом отсекаются. Это значит, что аргумент 12 имеет результатом "14" — другими словами, символьную строку из двух символов, первый из которых 1 (ASCII (31), 49 десятичное) и другой 4 (ASCII (34)), 52 десятичное). Максимальная длина строки — результата OCT$ ( ) — шесть символов.

Смотрите BIN$( ) и HEX$( ) для дополнительной информации, особенно об обработке беззнаковых и знаковых аргументов — метод тот же.

Использование: Эта функция форматирует вывод, преобразует тип, обеспечивает строковые операции и арифметику смешанных типов.

Пример:

```
PRINT OCT$ (&HFFFF)
177777
OK
```

См. также: BIN$ ( ), OCT$ ( ), STR$ ( ), VAL( ), Глава 2 о типах данных

оператор

ON ERROR GOTO

Цель: Установить обработку ошибок вашей собственной подпрограммой

Формат: ON ERROR GOTO *номер строки*

номер строки :: = любой существующий номер строки или 0 — отсутствие обработки

Действие: Этот оператор включает или останавливает обработку прерываний из-за ошибок. Он указывает, что подпрограмма пользователя может обрабатывать и пытаться исправить ошибку, которая возникла во время выполнения программы. Обработка прерывания по ошибке возникает, когда выдается какое-либо из сообщений об ошибках, перечисленных в приложении. Вместо выполнения прерванной программы управление передается строке программы, указанной в операторе ON ERROR, для обработки ошибки. Если указан 0, то обработка ошибок пользователем отключается, и производится нормальная обработка. В противном случае, обработка ошибок пользователем включена и начинается с указанного номера строки программы.

Номер строки начала обработки хранится в слове ONELIN (F6B9), и вы можете временно сохранить его — например, подпрограмма может временно преключить обработку ошибок, и может восстановить первоначальное значение при выходе.

Использование: Система обработки ошибок пользователем позволяет перехватывать почти все сообщения, генерируемые MSX-Бейсиком, и вести вашу собственную обработку. Вы должны выходить из программы обработки прерываний с помощью RESUME для продолжения нормальной работы, можно использовать ERR и ERL для определения в программе обработки прерываний места, где ошибка произошла.

Заметим, однако, что есть несколько ошибочных состояний, вызывающих немедленную стандартную (не пользовательскую) обработку — например, неверный тип данных в INPUT. ON ERROR не влияет на эти типы ошибок. Если ошибка указана в приложении, она может обрабатываться пользователем. Заметим, что обработка ошибок пользователем действительна и в непосредственном режиме — вы можете вызвать автоматический переход GOTO или, выполнив ON ERROR GOTO или и затем прямо введя команду, которая вызовет обрабатываемую ошибку.

Пример:

```
100 PRINT "нажмите Control-STOP для выхода"
110 ON ERROR GOTO 200
120 LLIST
130 GOTO 220
200 IF ERR = 19 THEN PRINT "печать остановилась"
205 GOTO 220
210 PRINT "на строке"; ERL;: ERROR ERR
220 ON ERROR GOTO 0 'конец обработки, возврат к нормальной
    работе
```

См. также: ERR, ERL, ERROR, RESUME

Цель: Оператор выбора — переход в зависимости от значения выражения

Формат: ON *арифвыр* {GOSUB / GOTO} *последовательность номеров строк*

последовательность номеров строк :: = один или несколько существующих номеров строк программы
арифвыр :: = любое разрешенное арифметическое выражение от 0 до 255

Действие: Этот оператор позволяет записать столько номеров строк, на сколько хватит места. Численное выражение вычисляется, и если оно не удовлетворяет ограничениям, выводится сообщение "недозволенное обращение к функции". Выполнение продолжается со следующего оператора. Если выражение ссылается на один из номеров строк в операторе (считая слева, 1,2 и т.д.), то этот номер строки используется для оператора GOSUB или GOTO.

Если выполняется ON GOSUB, то это равнозначно GOSUB номер-строки, и RETURN вернет выполнение программы к оператору, стоящему после ON GOSUB.

Использование: Этот оператор удобен, когда выполнение одной из нескольких различных ветвей определяется значением переменной или выражения. Если эти ветви короткие и прямо связаны с программой, используйте ON GOTO. Если ветви имеют сложную логику и могут быть оформлены как модули, вызываемые из многих точек, то используйте ON GOSUB и выполняйте каждое отдельное действие подпрограммой. Для избежания выходящих за границы величин, вызывающих ошибки, и управления вариантом "иначе", попробуйте

ON xx GOSUB 100, 200, 300, 400: GOTO *'уход* также.

Вы можете использовать этот способ для анализа данных, вводимых с клавиатуры:

```
10 ON ASC (INPUT$(1)) GOTO 101, 102, 103,
   104 . . .
20 GO TO 10
```

Пример:

```
200 INPUT "ВАШ ВЫБОР (1—3)"; CH
270 ON CH GOSUB 310, 320, 330
220
310 REM попадаем сюда если CH = 1
315 RETURN
320 REM попадаем сюда, если CH = 2
325 RETURN
330 REM попадаем сюда, если CH = 3
335 RETURN
```

См. также: GOTO, GOSUB

Цель: Позволяет программе пользователя обработать определенные ситуации

Формат:

```
        INTERVAL = тики
        KEY
ON  { SPRITE } GOSUB  номер строки
        STOP                [ , номер строки ] . . .
        STRIG
```

тики :: = интервалы в 1/60 секунды, 1—65535
номер строки :: = любой допустимый существующий номер строки

Примечание: только ON KEY и ON STRIG могут содержать больше одного номера строки — если какой-либо номер строки в этом списке пропускается, то все разделяющие запятые в списке сохраняются — в таком случае обработки прерывания производиться не будет.

Действие: Этот оператор определяет программу обработки событийных прерываний. Есть пять различных типов событий, но на самом деле возможны 16 событий: каждая функциональная клавиша может обладать собственной программой обработки, и возможны три триггерных входа. Когда эти "асинхронные" события случаются, выполнения программы прерывается по окончании исполнения текущего оператора, происходит GOSUB *номер строки*, и неявно активируется оператор STOP относительно данного события. RETURN в конце программы обработки возвращается к месту, где выполнение было прервано; происходит также разрешения события и запоминается следующее прерывание того же вида, и вызывается программа обработки прерывания после ее завершения (RETURN).

Первый вид прерывания — INTERVAL. Временной период — в тиках — должен быть указан. (Не делайте его слишком коротким, иначе вы

никогда не вернетесь из программы обработки, вызывая все виды ошибок). Максимальный возможный интервал — 65535/60, или 21 минута 50,7 секунд.

Второй — KEY. Вы можете указать до 10 номеров строк подпрограмм обработки в ON KEY. Только те из них, что действительно заданы обеспечивают обработку прерываний от функциональных клавиш.

Третий — SPRITE. Когда "сталкиваются" спрайта, то есть какие-нибудь части их максимальных возможных размеров накладываются — происходит прерывание. Вы должны использовать переменную в ваших операторах PUT SPRITE для сохранения следа последнего движения спрайта-прерывание всегда вызывается PUT SPRITE (или эквивалентным ему VPOKE).

Четвертый — STOP. Он позволяет обрабатывать прерывания при попытке остановить выполнение с помощью клавиш Control-STOP. Он не учитывает простое нажатие STOP, которое приостанавливает выполнение.

И, накомец, STRIG может определить обработку прерываний от трех различных триггеров джойстика на три различных номера строки программы: первый триггер — пробел, второй — от джойстика 1, третий — от джойстика 2.

Ни один из этих операторов не начинает обработку ошибок по прерыванию, и не отслеживает какие-либо события. Они просто связывают номера строк программы с типами событий (и позволяют указать временной интервал). Для каждого из возможных прерываний есть оператор типа *событие* ON, OFF и STOP для управления возможностью прерывания. См. описания этих операторов.

Проверка указанного номера строки происходит во время выполнения оператора. Весьма возможна ошибка "Несуществующий номер строки". Кроме того тип событий игнорирует существование других типов событий — то есть, если вы не хотите, например, чтобы обработка KEY — прерываний начиналась во время вашей обработки INTERVAL — прерывания, указывайте это операторами KEY(n) STOP в начале других (скажем, INTERNVAL) программ обработки прерываний.

Использование: Эти события — имеются в виду нажатия клавиш, движения спрайтов и истечение интервала времени — таковы, что могут случиться в любом месте вашей программы и зависеть от времени. Описываемая особенность позволяет вам полностью игнорировать обработку этих событий в программе — затем вставить программу обработки прерываний, которая позаботится о них полностью независимо от главной программы. Таким образом можно:

— двигать спрайт по экрану с помощью ON INTERVAL;
— создать в программе несколько подпрограмм, управляемых ON KEY;
— случайным образом создавать новые "цели" с временным интервалов;
— проверять на "столкновение" спрайтов; начинать движение объекта по нажатию триггера;
— удерживать пользователя от выхода из программы.;

Обязательно проверяйте и изолируйте каждую программу так, чтобы побочные эффекты не вызывали проблем. Использование полдюжины прерываний может полностью завести в тупик любого, пытающего отладить программу со странным поведением.

ON event GOSUB

Пример:

```
100 ON KEY GOSUB 4000, 4010, 4020, 4030, 4040 , , , 4070
110 FOR X=1 TO 5 : KEY (X) ON : NEXT
120 ON INTERVAL = 100 GOSUB 4050 'звуковой эффект
130 ON STRIG GOSUB 4060 'пробел
140 ON STOP GOSUB 4090 STOP ON 'игнорировать клавишу STOP
150 GOSUB 5000 'инициализировать игру
160 INTERVAL ON 'начать гудки каждые 2 секунды
```

См. также: PUT SPRITE, все операторы типа *событие* ON/OFF/STOP.

OPEN

Цель: Назначить устройство В/В буферу файла и инициализировать его

Формат:

OPEN "*имя устройства* [*имя файла*]"
[FOR { APPEND / INPUT / OUTPUT }] AS [#] *номерфайла*

имя устройства :: = CAS: CRT: GRP: LPT:
имя файла :: = любое строковое выражение, задающее имя файла
номерфайла : : = арифметическое выражение, задающее разрешенный номер файла между 0 и MAXFILES

Действие: Этот оператор начинает процесс В/В файла. Имя устройства проверяется на правильность: Для MSX Бейсик именами устройства являются CAS, CRT, GRP и LPT, и все они являются устройствам вывода (кроме CAS). Если имя устройства не воспринимается или имя файла неверно, происходит ошибка "Неверное имя файла". Если *номерфайла* был уже открыт (OPEN), происходит ошибка "Файл уже открыт". В остальных случаях *номер файла* назначается устройству, и выполняется необходимая инициализация. Например, для CAS: программа обработки автоматически начнет поиск имени файла — см. в BSAVE и BLOAD. Атрибуты INPUT (ввод) и OUTPUT (вывод) указывают направление потоков данных, а APPEND (добавить) —атрибут для расширенного Бейсика, встроенные устройства не могут его использовать — его наличие указывает, требуется операция последовательного доступа.

CAS — устройство последовательного доступа, устройство В/В в формате ASCII , обрабатывающее блоки данных, автоматически создаваемые MSX-Бейсиком. При работе с файлом символ control-Z (код 1A) обозначает конец файла и не

должен выводиться в файл — он закроет доступ к любым данным, выведенным (PRINT) после него.

CRT — устройство вывода, посылающее символы на экран. Работает только при SCREEN 0 и 1. Явное отличие от простого PRINT заключается в обработке запятых — CRT пропускает по 14 пробелов между началом печатных зон, независимо от того, требуется или нет свертка: PRINT пытается размещать данные в столбцах и переходит к следующей строке, если на этой не хватает места для полной печатной зоны.

GRP — устройство вывода, передает данные на графический экран — SCREEN 2 или 3- и использует матрицы 8×8 для символов.
Все символы при выводе затирают прежнее содержимое экрана — включая ранее выведенные символы. Чтобы уничтожить строку символов, либо нарисуйте прямоугольник и заполните его цветом, либо выведите (PRINT) строку на прежнее место после переключения цветов на пртивоположные оператором COLOR. Вывод пробелов не меняет абсолютно ничего.

LPT — устройство вывода, передает данные на MSX-устройство печати. LPRINT равнозначно PRINT#1 после открытия PRINT#1 как LPT:.

Использование оператора OPEN не только начинает B/Bb, но и позволяет также использовать номер файла во многих функциях — INPUT$( ), LINE INPUT, PRINT USING и т.д.

Использование: Оператор OPEN начинает любой процесс B/b. Вы должны выполнить его до использования любого процесса, связанного с файлами B/B. Команды группы LOAD/SAVE (включая MERGE и RUN) работают по умалчанию с номером файла 0 — вам не приходится заботиться об этом, но вы не можете открыть файл с номер-файла = 0 и ожидать, что он останется неприкосновенным.

Так как открытый БУФ (блок управления файлом, включая 256-байтовый буфер данных) — единственное, что автоматически сохраняется между двумя программными сегментами (при использовании LOAD, RUN), то это единственный путь передачи данных между двумя программными модулями. Изучите распределение памяти в приложении, чтобы понять, где и как это сделать.

Пример:

```
100 SCREEN 2 : OPEN "GRP:" FOR OUTPUT AS#1
110 PRINT#1,"График годового дохода от инвестиций:"

100 OPEN "CAS:" FOR INPUT AS #1
110 LINE INPUT # , CS$
```

См. также: Глава 4, секция о B/B; CLOSE, все виды INPUT и LINE INPUT, PRINT и PRINT USING, EOF ( ), MAXFILES=, INPUT$ ( ).

OUT

оператор

Цель: Посылает байт данных в порт вывода микропроцессора Z-80

Формат: OUT *адрес, данные*

адрес :: = арифметическое выражение, дающее целый результат
данные :: = арифметическое выражение, дающее результат в диапазоне 0—255

Действие: Адрес и данные вычисляются. Затем данные пересылаются в порт вывода процессора Z-80.

Использование: Если вы детально знаете MSX-спецификации, оператор OUT удобен для многих целей. Например, в PPI порт C по адресу (&HAA) управляет свечением индикатора "CAPS". Оператор OUT также полезен при разработке желающими специальных аппаратных средств: запомните, порты с (&H80) по (&HFF) зарезервированы под MSX — стандарт, с (&H00) по (&H7F) для изготовителей, с (&H100) до (&HFFFF) — свободны.

Примечание: Эти номера портов могут измениться в будущем, потому не используйте этот оператор в программах, предназначенных для тиражирования.

Пример:

```
100 OUT & HAA, INP (&HAA) AND & HBF 'включить индикатор
    "CAPS"
200 OUT&HAA, INP (&HAA) OR &H40 'выключить индикатор "CAPS"
```

См. также: INP ( ), приложение F

фукция

PAD

Цель: Ввод данных с Графического планшета MSX

Формат: *x* = PAD (*выбор*)

выбор :: = любое арифметическое выражение, дающее результат в диапазоне 0—7

Действие: Эта функция предназначена для ввода координат с графического планшета, подключенного к порту 1 или 2 для джойстика, и использующего дискретные импульсы для считывания с планшета. Значение *выбор* определяет, что вы хотите узнать:

Значение выбор	Джойстика порт 1	Джойстика порт 2
Считать блокнот (выдает-1, если есть прикосновение, иначе 0)	0	4
выдает-координату после чтения	1	5
выдает-координату после чтения	2	6
Считать состояние кнопки (выдает-1, если нажата, иначе 0)	3	7

Заметим, что при выполнении PAD (0) одновременно считываются значения, которые вы получите с помощью PAD (1) и PAD (2); то же касается PAD (4), PAD (5) и PAD (6). Так что, когда PAD (0) или 4, возвращает TRUE (-1) и x, y-координаты считываются и хранятся, пока вы позже не затребуете их. Ошибка "Недопустимое обращение к функции" происходит, если *выбор* во время вызова равен чему-либо отличному от перечисленного выше.

Использование: С помощью этой функции вы можете получить соответствующую информацию от аппаратных средств прямо из Бейсика, минуя сложные драйверы, PEEK или INP ( ) вызовы.

```
100 IF PAD(0) THEN PAD (1) : Y=PAD(2): GOSUB
    1000
105 GOTO  100 'ожидает ввода точки, затем
    обрабатывает (x, y).
```

Вычерчивание последовательности отрезков

Пример:

```
50  SCREEN 2
100 IF NOT PAD(0) THEN 100
110 IF PAD (3) THEN PSET (PAD(1), PAD(2)) : GOTO 100
120 LINE – (PAD(1), PAD (2)): GOTO 100
```

См. также: PDL ( ), графические операторы

Цель: Заполняет область графического экрана сплошным цветом

Формат: PAINT [@] [STEP] (*x*, *y*) [, *цвет*] [, *граница*]

x :: = любое арифметическое выражение, задающее — координату, 0—255

y :: = любое арифметическое выражение, задающее — координату, 0—191

цвет :: = любое арифметическое выражение, задающее цвет закраски; 0—15; если опущено, закрашивается текущим цветом

граница :: = любое арифметическое выражение, определяющее цвет границы области закраски; 0—15; если опущено, используется текущий цвет изображения. Этот параметр не применим в режиме SCREEN 2.

@ :: = ничего не значит, только для удобочитаемости

STEP :: = указывает, что точка определяется относительно последней выведенной точки

Примечание: Если *цвет* опущен, но *граница* присутствует, то перед *границей* должны стоять две запятые, указывая отсутствие цвета.

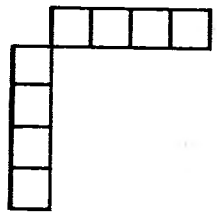
Действие: Оператор используется для закрашивания области графического дисплея в режиме SCREEN 2 или 3. Область для закрашивания должна быть ограничена одним цветом, и она будет полностью закрашена единственным цветом. Если цвет закрашивания не совпадает с цветом границы, цвет *границы* должен быть указан. Если цвет закрашивания совпадает с текущим цветом изображения, установленным оператором COLOR, то вы можете опустить атрибут цвета. Точка начала закраски может быть любой точкой внутри области (но не на границе); вы можете указать абсолютный адрес

точки или, атрибутом STEP, указать смещение от последней точки, указанной вами. Однако, точка должна быть внутри текущего дисплейного окна, иначе произойдет ошибка "Недопустимое обращение к функции".

Использование: Это не скорейший путь закрасить объект — атрибут BF оператора LINE в несколько раз быстрее. Заметим, что закраска всего экрана занимают около 15 секунд, но небольшие объекты обычно закрашиваются намного быстрее. Один метод использования — добавить быстрый "BF" атрибут к CIRCLE:

```
CIRCLE (x, y) . . . : PAINT STEP (0, 0)
```

Заметим, в отличие от других графических команд, координаты в PAINT должны быть внутри экрана 0:255, 0:191. PAINT также проверяет наличие границ только по вертикали и горизонтали — и объект признается ограниченным, если ограничены вертикаль и горизонталь. Наконец, запомните, что даже мельчайшая прореха в границе позволит оператору PAINT "пролиться" наружу и уничтожить большие области на экране.



Пример:

```
DRAW A$ : PAINT (105, 22), 3, 8 (если (105, 22) внутри неровной
фигуры, созданной DRAW с цветом 8, что закрасить всю ее цветом 3.)
```

См. также: COLOR, LINE, CIRCLE, DRAW, PSET, PRESET
Глава 4 о графике

Цель: Ввод кода положения игрового рычажка

Формат: *x* = PDL (*выбор*)

выбор :: = любое арифметическое выражение
1—12

Действие: Функция считывает код положения рычажка. К любому порту джойстика можно подключить только один рычажок, и считывание производится при помощи дискретных импульсов. Операция считывания требует некоторого времени — около 0,01 секунды. Если *выбор* нечетный, считывается порт 1, если четный — порт 2. Если *выбор* не удовлетворяет ограничениям, появляется сообщение "Недопустимое обращение к функции".

Использвание: Функция PDL ( ) упрощает ввод данных, с некоторых специфических типов устройств в Бейсик-программе. Обычно PDL (1) и PDL (2) используются для чтения портов 1 и 2. Если нет подключенного рычажка, функция вернет значение 255, если есть — то значение от 0 до 254.

Пример:

```
50  SCREEN 2
100 PSET (PDL(1), PDL(2)): GOTO 100
```

См. также: PAD ( ), STICK ( ), STRIG ( )

Цель: Позволяет проверить любую ячейку в памяти

Формат: *x* = PEEK (*адрес*)

адрес :: = арифметическое выражение, дающее результат со знаком или без знака,
-32768 < = *адрес* < = 65535

Действие: Эта функция позволяет вам проверить любую ячейку памяти в адресном пространстве MSX — компьютера. Поскольку команды программ и Бейсик.-программы, их обрабатывающие, размещаются на страницах избранного по умолчанию, слота, то PEEK ( ) обычно не используется для прочерки чего-либо, кроме расширителя магистрали по умолчанию (первичный или вторичный расширитель магистрали 0, плюс возможный расширитель на 16К ОЗУ в другом расширителе магистрали).

Аргумент этой функции может рассматриваться как знаковое или беззнаковое число; смотри в BIN( ) детальное описание.

Использование: Лучшее использование этого оператора — проверив предворительно версию вашего MSX ПЗУ (PEEK (43) до PEEK (47), все нули в версии 1.0) экспериментировать с данными, хранимыми в ОЗУ. Вы можете использовать PEEK ( ) для проверки значений всех переменных Бейсика (начинающихся с (F380) и использующих остаток памяти).

Пытаясь прочитать двухбайтовую переменную (16-битное слова), помните, что процессор меняет расположение байтов в слове памяти. Например, чтобы прочесть слово по адресу (&HF9FE), используется оператор:

```
PRINT PEEK (&HF9FE) + 256 *PEEK (&HF9FF)
```


Пример:

```
100 PRINT "Введите 16-битный адрес (добавьте для слова —) :";
110 LINE INPUT AD$: AD=VAL ("&H"+AD$): PK = PEEK (AD)
120 IF INSTR (AD$, "—") THEN PK = PK+256*PEEK (AD+1)
130 PRINT LEFT$ ("000"+HEX$(PK), 4): GOTO 100
```

См. также: POKE

Цель: Создание музыкальных эффектов

Формат: PLAY [*стрвыр*][, *стрвыр*][, *стрвыр*]

стрвыр :: = любое разрешенное строковое выражение, содержащее программу на музыкальном Макро Языке (ММЯ)

Действие: Эта команда испольдует Музыкальный Макро Язык, или ММЯ. Вы можете указать, чтобы три (или меньше) музыкальных голоса збучали одновременно. Более того, вы можете создать, "подпрограммы", которые можно вызывать из команд ММЯ. Используя этот язык, вы можете перевести почти любую музыкальную фразу. Кроме того, есть ограниченные возможности выстраивать збуки "в очередь" и исполнять другие операторы Бейсик, пока идет збукобой фон.

ММЯ состоит из одной, двух или трех строк (для одного, двух или трех голосов), использующих "ключевые" командные буквы; буквы можно разделять запятыми для улучшения читаемости. Строки следуют за оператором PLAY и могут быть получены в результате любых строковых операций. Некоторые команды ММЯ должны заканчиваться точкой с запятой (;), ASCII(3b), чтобы избежать ошибок при интерпретации. Каждая команда одно-буквенная, с возможным добавлением, после которого часто следует один численный параметр. Этот параметр можно задавать переменной, используя следующий формат:

команда = переменная;

Если для одного из трех голосов указана нулевая строка, то этот канал останется молчащим. Далее следует краткое описание возможностей ММЯ.

C D E F G A B
до ре ми фа соль ля си

Это основные ноты, доступные, по порядку, снизу вверх. Буквы вызывают исполнение соответствующей ноты в текущей октаве текущей длительности и формы.
Ноты можно модифицировать добавлением

\# + — . *длительность.*

Знаки # и + указывают диез, знак — обозначает бемоль. Запомните, C — соответствует B и B+ соответствует C, всегда той же октавы. После добавлений вы можете указать длительность ноты, если она нестандартная, — либое число от 1 до 64. В любом случае, 1= целая, 2= половина. 4 = четверть и т.д. Длительность не может задаваться с помощью переменной. Наконец, после всего этого вы можете добавить любое число точек (.), каждая из них удлиняет ноту на половину.

O *число.*
Выбор/новой/октавы. Наименьшая октава 01, наибольшая 08 (C — наименьшая, а B — наибольшая ноты в октаве). Вы можете указать новую октаву в любой момент, когда вам потребуется нота вне текущей. 04 установлена по умолчанию (соответствует первой октаве пианино). Установка октав неизменна до следующего 00 или до сброса оператором BEEP.

N *число.*
Вы можете задать любую ноту числом. Самая низкая нота – N1, самая высокая N96 (заметим, что До первой октавы— это O4C= 36).
Запомните, доступна каждая нота, т.е. N37 = C+, а не D. Использование формы с переменной —

N = *переменная*; – эффективный способ для сохранения музыкальных данных. Если вы желаете сохранить нестандартной длительности, вы должны записать точку с запятой (;) после номера точек (или после переменной) и затем длительность.

L *длина*.
Устанавливается длительность по умолчанию для последующих нот (умолвание — 4).
Можно также задать длительность носты с помощью переменной. Длина возможна в границах 1 — 64, и действительная длительность определяется как (1/длина AND текущая установка темпа). 1 = целая, 2 = половинная, 4 = четверть и т.д. Заметим, что C32 равносильно L32C, но L влияет и на все последующие ноты до его сброса (BEEP) или другого L.

R *длительность*.
R – специальная молчащая нота, не генерирует никакого звука. Вы можете модифицировать *K* тем же путем, как, например, C. Длительность по умолчанию 4.

T *темп*.
Устанавливает темп исполнения музыки. Темп может быть от 32 до 255 (BEEP сбрасывает темп на 120 по умолчанию).

V *громкость*.
Устанавливает громкость, по умолчанию 8, возможны значения от 0 до 15, неизменна до следующей V (или сброса оператором BEEP).
V сбрасывает S и M на их значения по умолванию.

M *нч*.
Установка несущей частоты звукового генератора. *Нч*. может быть в пределах от 1 до 65535, по умолчанию 255, BEEP сбрасывает и ее. Детально смотри в SOUND.

S *форма*.
Устанавливает форму волны. По умолчанию = 1, может быть от 1 до 15, сбрасывается только BEEP или другим S. В основном используется вместе с M для эффектов: чистые обертоны при 1,4,8,10,11,12,13,14. Детально смотри в SOUND (не используйте V при установке S).

X *строковая переменная*.
Выполняет "подпрограмму" на ММЯ, содержащуюся в строковой переменной. Возможно любое необходимое вложение.

Исключая отмеченные места, константы можно заменять переменными, в кочестве разделителей можно использовать точки с запятой или пробелы, или не разделять ничем, BEEP (или любая ошибка) сбрасывает все параметры на значения по умолчанию и прекращает выполнение PLAY.
Когда выполняется PLAY, все строки обрабатывается и преобразуются во внутренний музыкальный формат, организованный в три очереди (по одной для каждого голоса). Эта операция выполняется быстро, и, как только она завершается, исполнение программы продолжается со следующего оператора. Тем временем PSG автоматически воспроизводит одну, две или три ноты одновременно с выполнением вашей программы, продолжая, пока очереди не опустошатся (вы можете контролировать их функцией PLAY( )). Только BEEP или ошибка остановят музыку. Вы сможете выполнять последовательный PLAY, они просто добавят больше музыкальных нот к очереди. Однако, если при выполнении PLAY очереди заполнятся до предела, выполнение программы приостанавливается, пока не будет сыграно достаточно музыки, чтобы сгенерированная вся музыка уместилась в очереди.

Использование: При достаточном упорстве любая партитура из трех или меньше частей может быть переведена на ММЯ. И если вы организуете ваши операторы PLAY аккуратно, вы можете играть музыку, пока выполняются другие части вашей программы — еще одна фоновая задача. Одна проблема: невозможно определить, когда очередь почти пуста — вы можете только узнать, когда она опустошилась. Более, два последовательных оператора

PLAY "C" :PLAY "C"

всегда вставляют короткий промежуток между концом музыкальных данных одного оператора PLAY и началом следующего множества данных. Примечание. Если вы продолжаете пересылать данные в очереди последовательными PLAY, расположенными так, чтобы музыка не останавливалась, то в исключительных случаях очереди могут рассинхронизироваться.

Один способ сформировать заметный промежуток нотами показан ниже:

```
10 A$ = "M2000S11": PLAY A$, A$, A$
```

что позволяет добавить оттенок "стаккато" к звуку (или попробуйте M1000). Однако, использовав S, вы потеряете весь контроль громкости с помощью V.

Пример:

```
100 ON ERROR GOTO 110: GOTO 120
110 RESTORE: RESUME 120 'вернуться к началу данных на ММЯ для
    PLAY
120 GOSUB 1000 ' рисует основной экран, просит нажать клавишу
130 READ A$, B$, C$: PLAY A$, B$, C$
140 GOSUB 1090 ' выводит вопросительный знак
150 IF INKEY$ < > " " THEN 170
160 IF PLAY (0) THEN 160 ELSE 130
170 BEEP ' выключить музыку и продолжить дальше
180 DATA m 2000 s11t 200,m2000 s11t 200,m 2000 s11t 200
181 DATA c2.de2.ce2c2e1, o3c2o2g2o3c2o2g2, r1r1r1r1
```

См. также: PLAY ( ), SOUND

PLAY

фукция

Цель: Определяет, есть ли еще ждущая в очереди несыгранная музыка

Формат: *x* = PLAY (*голос*)

голос :: = арифметическое выражение, 0—3, голос для проверки

Действие: Функция сообщает, есть ли музыка в очереди и в процессе игры в качестве фоновой задачи. PLAY (0) сообщает, есть ли еще несыгранная музыка, PLAY (1) проверяет только голос 1, PLAY (2) — голос 2 и PLAY (3) — голос 3. Все функции выдают 0 FALSE, если музыка не играется, если же что-нибудь есть в очереди и играется, выдают : -1 (TRUE).

Использование:

```
10 PLAY A$, B$, C$
20 IF PLAY (0) THEN 20
```

(выполнение программы не продолжится, пока музыка не будет сыграна).

Вы можете в вашем "главном" цикле, использовать эту функцию для определения останова музыки, затем продолжить ее:

```
IF NOT PLAY (0) THEN READ A$ : PLAY A$
```

Пример:

```
1000 'подпрограмма, вызывающая столкновение
1010 IF PLAY (3) THEN BEEP 'не может быть музыки в очереди 3
1020 PLAY " ", " ", CX$ 'остановить музыку
1030 'сейчас приближается остановка . . .
```

См. также: PLAY, SOUND, BEEP

фукция

POINT

Цель: Выдать код цвета точки на графическом экране

Формат: *z* = POINT (*x*, *y*)

x :: = любое арифметическое выражение, дающее -координату, 0—255.
y :: = любое арифметическое выражение, дающее -координату, 0—191.

Действие: Эта функция в графическом речелселе выдаст код цвета указанной точки на экране-число от 0 до 15 (смотри список в COLOR). Запомните, в SCREEN 2 восьмь последовательных точек в ряду окращиваются одним цветом. Если точка за пределами координатных ограничений, будет выдана -1; если точка не принадлежит избражению, будет выдан код цвета фона.

Точнее, если точка — точка иображения (бит = 1), выдается цвет изображения, если это точка фона — выдается цвет фона и если она вне экрана, выдается минус один (-1). Не делается никаких проверов на неправильное использование (попытку вызова POINT в SCREEN 0 или 1).

Использование: Одно из назначений POINT — определить, принадлежит ли точка изображению. Другое — это определение цвета границы для PAINT. Можно использовать POINT для "следования" за рисунком на экране — определения, не было ли столкновения с объектом, нарисованным на графическом экране, или "отскакивания" от стенки и т.д.

Пример:

```
100 GOSUB 1000 'передвигает, обновляет X0 и Y0
110 IF POINT (X0, Y0) = 4 THEN 100 'удара не было
120 GOSUB 200: GOTO 500' столкновение произошло
```

См. также: PSET, PPESET, COLOR, Главу 4 о программировании графики.

POKE

оператор

Цель: Записывает данные в любую ячейку в адресном пространстве памяти

Формат: POKE *адрес, данное*

адрес :: = арифметическое выражение, дающее беззнаковое или знаковое число от —32768 до 65535
данное :: = арифметическое выражение, 0—255

Лейсивие: Этот оператор вычисляет адрес, данное и сохраняет данное по адресу — один байт. Адрес может быть знаковый или беззнаковый; смотри в BIN$ ( ) детальное объяснение. Если адрес не удовлетворяет ограничениям, происходит ошибка "Переполнение", если данное — "Недопустимое обращение к функции". POKE обычно не используется ни для чего, кроме записи данных в старших 32К ОЗУ, начиная с (H8000).

Использование: Вы можете использовать POKE для модификации вашей программы, или изменения указателей либо величин, хранимых в памяти. Вы можете разместить в ОЗУ программу на машинном языке, записав ее побайтно. Этот оператор позволяет вам экспериментировать с "подвалом" компьютера.

Примечание: Неверные данные или адрес в POKE могут вызвать нежелательные результаты.

Пример:

```
100 POKE &HF3B0, 20
(Устанавливает ширину (WIDTH) SCREEN 1 равной 20 без очистки)
```

См. также: PEEK, USR ( )

фукция

POS

Цель: Выдает позицию курсора в строке (номер столбца на текстовом зкране)

Формат: *x* = POS (*аргумент*)

аргумент :: = все, что угодно — полностью игнорируется функцией

Действие: Функция выдает с номер столбца курсора на последнем текстовом экране — SCREEN 0 или 1. Это значит, что если вы перешли к SCREEN 2 от SCREEN 1 и вызвали POS (0), вы получите позицию курсора в строке, для SCREEN 1. Возвращаемая величина — между 0 и (WIDTH-1).

Использование: Эта функция, вместе с CSRLIN выдает текущее положение курсора. Вы можете определить в подпрограмме, где курсор находится, передвинуть его для обновления части экрана, и вернуть назад с помощью LOCATE перед выходом из подпрограмм.

Пример:

```
1000 X = POS (0): Y = CSRLIN
1010 LOCATE X1, Y1
1020 PRINTZ$;
1030 IF POS(0) > 28 THEN PRINT ELSE PRINT "<—"
     .....
1100 LOCATE X, Y : RETURN
```

См. также: CSRLIN, LOCATE, WIDTH, SCREEN

Цель: Перевод точки графического экрана в цвет фона.

Формат: PRESET [@] [STEP] (*x*, *y*) [, *цвет*]

x :: = арифметическое выражение, задающее x-координату (колонку)
y :: = арифметическое выражение, задающее y-координату (строку)
Замечание: обе координаты должны иметь значение в диапазоне -32768=< nn< = 32767
цвет :: = арифметическое выражение со значением 0—15
@ :: = для удобства чтения
STEP :: = задает точку по отношению к последней использованной графической точке.

Действие: Вызывает сообщение об ошибке "Illegal function call", "Неверный вызов функции" в случае, если в операторе SCREEN не были заданы параметры 2 или 3. В противном случае переводит цвет заданной точки в цвет фона (то есть устанавливает бит в 0, отмечая таким образом, что точка имеет фоновый цвет). Формат задания координаты поясняется при описании CIRCLE. Если задан параметр *цвет*, этот оператор выполняется как PSET: соответствующий бит устанавливается в состояние цвета изображения, цвет изображения устанавливается заданным (не фоновым).

Использование: PRESET (и PSET), без параметра *цвет*, позволяют установить и сбросить цвет любой точки из цвета изображения в фоновый цвет. Этот оператор при отсутствии параметра *цвет* обеспечивает способ изменения VRAM без изменения таблицы цветов — изменяется лишь таблица шаблона. Этот оператор может быть также использован для селективного стирания изображения с экрана. Кроме того, использование этого оператора вместе с PSET позволяет установить графический аккумулятор без рисования на экране.

Пример:

```
1000 C=POINT(X,Y): PRESET (X,Y)
1010 IF POINT (X,Y) <> C THEN PSET (X,Y), C
1020 'установить графический курсор в точку без стирания.
```

См. также: PSET, POINT ( ), LINE, CIRCLE
Глава 4, графика.

Цель: вывод данных на экран (печатающее устройство).

Формат: [L] { PRINT / ? } [# *ном. файла,*]
[USING *формат;*] *список вывода.*

Ном. файла :: = номер любого открытого файла, 0—15, в случае LPRINT — недопустимо.
Формат :: = строковое выражение, содержащее информацию о формате.
Список вывода :: = список выражений любого типа, разделенных запятыми или точкой с запятой; список может быть пуст.
L :: = задает вывод на печатающее устройство; задание # ном. файла при этом недопустимо.
Замечание: если # ном. файла задан, то запятая перед списком вывода не пишется. Кроме того, оператор PRINT для сокращения записи может быть заменен знаком "?", что недопустимо для LPRINT.

Действие: Оператор PRINT посылает данные на устройство вывода. Обычно это экран, однако этот оператор позволяет выводить данные на печатающее устройство, в любой файл или на любое устройство вывода, для которого был выполнен оператор OPEN.
Выводимые данные могут быть представлены выражениями любого типа или отсутствовать совсем. Выражения, естественно, могут быть переменными или константами; в остальных случаях они вычисляются и результат выводится на дисплей. MSX-Бейсик допускает автоматическое форматирование; кроме того, используя параметр USING, можно осуществлять форматирование по усмотрению пользователя. Оператор PRINT выводит данные только на текстовый

экран — т.е. SCREEN 0 или 1.

При использовании автоматического форматирования, распределение элементов данных на экране осуществляется соответственно разделителям между ними. Разделителем являются запятая (,) или точка с запятой (;). Запятая указывает, что элементы данных, которые она разделяет, должны юстироваться влево, в пределах 14-символьной зоны вывода оператора PRINT; недостающие позиции справа заполняются пробелами. В результате этого данные располагаются в колонку. (Оператор WIDTH является значащим только при выводе на экран: после вывода первой зоны в случае нехватки места для полной следующей зоны в строке, курсор перемещается на начало следующей строки без вывода пробелов). Если данные выходят за конец зоны, следующая зона располагается прямо за концом данных без вывода пробелов — однако, на экране в строке не может уместиться более 2-х зон (3×14>40). В случае разделения элементов данных точкой с запятой они располагаются непосредственно друг за другом без разделения пробелами.

Если после последнего элемента данных стоит запятая или точка с запятой, курсор остается в той же строке после соответствующего пробела; в противном случае выводится пара символов окончания строки — 0D, 0A. (На экране это приводит к переводу курсора в начало следующей строки). Если в списке встречаются лишние запятые, соответствующие пустым элементам данных, пропускается соответствующее число зон печати.
В некоторых случаях точка с запятой может быть опущена — например, в случае присваивания литеральной строки переменной, за которой следует другая строка, заключенная в кавычки — однако, это делать не рекомундуется, посколь-

ку отдельные элементы должны быть разделены.

По умолчанию форматирование числовых данных осуществляется следующим образом: прежде всего выводится пробел. Если число отрицательно, выводится знак "—", в случае положительного числа выводится пробел. Если возможно, Бейсик избегает вывода на экран всех десятичных позиций и выводит их в экспоненциальном виде (например: 3.4E+14), когда десятичная точка выходит за 14-ю значащую цифру мантиссы. К концу числа всегда добавляется пробел. Строковые данные воспроизводятся без изменений. Параметр L задает вывод на MSX-печатающее устройство, а не на экран. Такой вывод осуществляется так же, как если бы пользователь открыл устройство "LPT:" для вывода и затем использовал PRINT с этим номером файла, но нет необходимости открывать или закрывать файл — более того, оператор LPRINT не допускает использования с параметром *#ном. файла*. Отдельная позиция колонки курсора, LPOS ( ), сохраняется для всех выводов оператором LPRINT.

Вывод может осуществляться на любое устройство, открытое оператором OPEN, для чего используется параметр *# ном. файла*. Необходимо помнить, что вывод происходит так, как если бы он осуществлялся на экран, байт за байтом. Более того, если надо вывести несколько числовых данных в строке и затем прочитать их оператором INPUT, в строке между элементами следует выводить запятые. В случае вывода в одну строку более одного строкового элемента, их следует при выводе заключать в кавычки — с помощью функции CHR$ (34) — до и после каждого элемента; благодаря этому оператор INPUT# сможет вводить эти элементы точно так, как они выводились. Пользователь может

открыть устройство "CRT:" и затем выводить на данные с помощью оператора PRINT #. Это все равно, что осуществлять вывод непосредственно с помощью оператора PRINT, за исключением того случая, когда ширина зоны печати не задана с помощью оператора WIDTH, а просто равна 14 символам. И, наконец, открытие устройства "GRP:" предоставляет очень ограниченный способ вывода на графический экран — SCREEN с параметром 2 или 3 — но при этом не осуществляется очистки области экрана перед добавлением следующего символа к существующему образу, и при вычислении зон WIDTH не опознается.

Параметр USING позволяет задать форматирование информации для элементов оператора PRINT. Формат строки может быть задан единожды, при этом Бейсик использует его для каждого элемента в операторе PRINT, начиная всякий раз с начала; формат строки можно задать для всей строки вывода, это позволяет вставлять поля в выводимый текст.

!
Первый символ соответствующего элемента строки данных должен печататься здесь.

\
Этот символ отмечает позиции первого и последнего символов, подлежащих выводу. Тогда количество символов извлекается из соответствующего элемента данных, дополняется справа пробелами и помещается в нужную позицию поля.

&
В эту позицию помещается один или более символов. Вся строка в соответствующей позиции вставляется в это место. Эта метка представляет любое количество пробелов в длине. Все после-

дующие элементы формата сдвигаются вправо, чтобы освободить достаточно места для строки.

#
Этот символ используется для указания на арифметическую цифру. Может быть задано до 24 цифр, а десятичная точка может либо присутствовать, либо быть опущена. Соответствующий элемент данных помещается в поле (в случае отсутствия невидимая десятичная точка предполагается справа); отсутствующие цифры слева от десятичной точки дополняются пробелами (число юстируется вправо); неиспользованные позиции справа от десятичной точки заполняются нулями. MSX-Бейсик обеспечивает максимальную точность в 14 цифр. Числа при необходимости округляются.

+
Плюс в начале или конце числового поля задает вывод знака — либо плюс (+), либо минус (-) именно в этой позиции. Ведущий знак перемещается в позицию непосредственно перед первой цифрой.

—
Знак минус помещается только в конце числового поля и вызывает печать знака в этой позиции только для отрицательных чисел; для положительных чисел печатается пробел.

**
Две звездочки в начале числового поля представляют две цифры (как если бы они были), но они приводят к тому, что все ведущие пробелы в поле, сгенерированные вследствие того, что число было короче поля, заполняются звездочками (в качестве "символов защиты от допечаток").

££
Эти два символа используются для определения двух числовых позиций аналогично звездочкам, но вызывают печать одного символа фунта слева от старшей значащей цифры.

**£
Эти символы вызывают печать одного символа денежной единицы (£ или $) слева, как в предыдущем случае, но в оставшихся позициях слева от него печатаются символы *, как в случае **.

,
Запятая непосредственно слева от десятичной точки представляется как любая цифра. Однако, она задает при выводе генерацию точек для разделения на группы по 3 цифры. Точка занимает одну цифровую позицию, что является ошибочным в случае экспоненциальной нотации (ниже).

^^^^
Эти четрые символа после числового поля вызывают печать числа в экпоненциальной (научной) нотации — добавляется E+dd или E-dd, где dd-две цифры экспоненты.

Есди число, подлежащее печати, больше, чем формат поля, или оно стало больше в результате округления, число печатается, однако перед ним выводится символ процента, чтобы предупредить пользователя.

Использование: Оператор PRINT в первую очередь предназначен для связи компьютера с внешним миром. Поскольку этот оператор так сложен, необходимо попробовать вывести несколько сомнительных элементов данных, чтобы увидеть результат его использования.

С помощью разумного использования оператора PRINT совместно с операторами LOCATE, вы можете осуществлять вывод в любое желаемое место экрана. Оператор PRINT никогда не очищает никаких строк на экране — он только выводит пробелы, как рассказано ранее. Единственным узким местом при выводе чисел с помощью оператора PRINT является неизбежное появление ведущих и замыкающих пробелов, но можно исправить дело, используя PRINT USING. Однако, PRINT USING не может выводить ведущий условный пробел для положительных чисел.

Если в результате использования единственного оператора PRINT не удалось добиться желаемого результата в выводимой строке (но не из-за неточности), тогда следует закончить оператор точкой с запятой и выполнить еще один оператор PRINT. Аналогичные хитрости работают и для вывода в файл.

Конечно, вы можете сделать вычисления для вывода "на лету" — запишите последнее выражение в оператор PRINT:

Пример:

```
PRINT 1; -2; 3
1 -2 3
OK
PRINT USING "#"; 1, 2, 3, : PRINT -4
123-4
OK
FM$="The & in Spain falls at ##.# % relative humidity."
OK
PRINT USING FM$; "rain", 5
The rain in Spain falls at 5.0% relative humidity.
(при 5% относительной влажности в Испании идет дождь — прим. пер.)
OK
PRINT , , , 3'b указывает на пробелы
bbbbbbbbbbbbbb
bbbbbbbbbbbbbb 3
OK
LPRINT A, "abcde"; zx;'','';YM;CHR$(13);
(выше приведена "хитрость" для принтеров, которые пропускают дополнительную строку при обнаружении перевода строки)
```

См. также: INPUT, Глава 2 "формат данных."

Цель: Установить цвет точки в цвет изображения на графическом экране.

Формат: PSET [@] [STEP](*x*, *y*) [,*цвет*]

x :: = арифметическое выражение, дающее x-координату (колонку)
y :: = арифметическое выражение, дающее y-координату (строку).
обе координаты должны находиться в диапазоне
-32768 = < nm < = 32767
цвет :: = арифметическое выражение, задающее цвет, в диапазоне 0—15; если этот параметр опущен, используется текущий цвет изображения.
@ :: = только для документирования, не оказывает никакого действия.
STEP :: = дадает точку относительно последней графической точки.

Действие: Если используется не графический экран — в операторе SCREEN нужны параметры 2 или 3 — то выводится сообщение об ошибке: "Illegal function call","Неверный вызов функции".
В случае графического экрана, если данная точка находится вне экрана (0: 255, 0: 191), оператор не оказывает никакого действия; если точка попадает в экран, соответствующий бит устанавливается в 1, индицируя цвет изображения, и соответствующий ему код цвета устанавливается равным цвету изображения, заданному последним оператором COLOR, или равным заданному параметром *цвет* в текущем операторе. (Для пояснения работы с точками см. LINE и CIRCLE). Если соответствующий точке бит уже находится в состоянии цвета изображения, то воздействие оказывается лишь на код цвета (и соседние точки).

Поскольку информация о цвете изображения используется для воздействия на 8 соседних

точек, то этот оператор может изменить цвет других точек, уже находящихся на экране в той же самой строке (в случае SCREEN 2). Поэтому, важно знать, находитесь ли вы на той же самой строке границы шириной в 8 точек экрана, что и другие точки, или нет. В случае SCREEN 3, каждый блок точек размером 4x4 создает на экране изображение одной точки, поэтому любое изменение в этом блоке вызывает изменение цвета всего блока — поэтому не существует битов цвета основного или фона изображения, а существует единственная тетрада цвета.

Использование: Помимо различных других действий этот оператор изменяет цвет графического курсора. Кроме того, этот оператор может быть использован для различных действий с точками экрана. Вследствие неизбежного побочного эффекта при использовании нескольких цветов, вы должны быть внимательны всякий раз при использовании этого оператора, чтобы при этом не изменить цвет уже нарисованной части изображения.

Пример:

```
100 SCREEN 2: FOR Y=1 TO 120: PSET (.2*Y^2, Y): NEXT
```

(изображает часть параболы).

См. также: PSET, POINT Глава 4 раздел графики.

Цель: Управление размещением и выбором спрайтов.

Формат: PUT SPRITE *спрном* [, [@] [STEP] (*x*, *y*)] [, *цвет*] [, *спробраз*]

x :: = арифметическое выражение, задающее x-координату (колонку)
y :: = арифметическое выражение, задающее y-координату (строку).
Обе координаты должны находиться в диапазоне от -32768 до 32767
спрном :: = арифметическое выражение в диапазоне от 0 до 31, задающее 1 из 32 спрайтов, подлежащих воздействию.
спробраз :: = арифметическое выражение от 0 до 63 или от 0 до 255 (в зависимости от последнего оператора SCREEN), задающее, какой из 256 образов спрайтов создается с помощью SPRITE$ (*спробраз*). Это связано с количеством спрайтов, которое можно вывести на экран.
цвет :: = это арифметическое выражение в диапазоне от 0 до 15 задающее цвет спрайта (предполагается цвет изображения)

@ :: = только для документирования, не оказывает никакого действия.
STEP :: = указывает, что точка задается относительно последней графической точки.
Замечание: Любой из параметров может быть опущен, но соответствующая ему запятая должна быть сохранена.

Действие: Этот оператор оказывает действие на спрайты, находящиеся на экране, а не на изображения, сохраненные с помощью функции SPRITE$ ( ). Этот оператор устанавливает связь образа со спрайтом, устанавливает его цвет и определяет местоположение. Допустимо получение 32 видимых спрайтов, однако, если на одну строку выводится более 4 спрайтов, только 4 из них с младшими номерами будут видимы, остальные

будут забланкированы на строках развертки (VDP(7) сообщит вам, о наличии "пятого" спрайта на любой строке развертки при его обнаружении). Для установки всех этих характеристик можно использовать оператор PUT SPRITE или с помощью запятых указать на отсутствие некоторых из характеристик; вы можете изменить только один из параметров, сохранив остальные неизмененнными.
Точка задается также, как и все графические точки; см. также описание операторов LINE и CIRCLE. Но в отличие от других операторов в операторе PUT SPRITE координаты точки *x* и *y* всегда вычисляются по модулю размера экрана — как в случае отрицательного числа, так и в случае слишком большого положительного числа. Учтите, что точка задает левый верхний угол спрайта. Значение координат точки может быть и отрицательным (незначительным по величине) или выходить за размеры экрана, помещая спрайт так, что он будет лишь частично виден на экране дисплея. Например, *y* = -1 поместит спрайт в самую верхнюю часть экрана; еще меньшее число выведет спрайт за границу экрана. Или же *y* может быть равен 191, и в этом случае спрайт полностью выйдет за границу экрана. Т.к. спрайт двойного размера (16X16) может иметь высоту размером в 32 точки, то он полностью выйдет за границу экрана при *y* = -33. Точно так же при *x* = 255 спрайт полностью выезжает за правую границу экрана, а *x* = 0 переводит спрайт в самое левое положение на экране. Если *x* будет отрицательным, то спрайт будет исчезать за левой границей экрана.

Вы можете также изменять цвет. По умолчанию при выполнении оператора SCREEN всем спрайтам присваивается цвет, совпадающий с цветом изображения, но вы можете изменить этот цвет, указав его явно.

Спрайты могут состоять из 8x8 точек (8 байтов, всего 256 изображений) или из 16x16 точек (32 байта, всего 64 изображения). Более того, размер спрайта во всех направлениях может быть удвоен. Этим управляет параметр "размер" в операторе SCREEN.
Наконец, вы умышленно можете накладывать спрайты друг на друга, создавая составной двухцветный спрайт. В этом случае действует следующее правило: спрайт с меньшим номером накладывается сверху. (Заметим, что в этом случае постоянно будет существовать условие столкновения "спрайтов — в этом случае нельзя использовать ON SPRITE GOSUB).

Использование: Спрайты облегчают вам создание движения / мультипликации средствами языка Бейсик. Для перемещения или изменения спрайтов используется оператор PUT SPRITE. Этот оператор может также изменять "изображение", используемое для конкретного спрайта, для достижения эффекта мультипликации. Не путайте это с функцией SPRITE$ ( ), которая только определяет изображение спрайтов. Любое из этих изображений (или несколько) может быть использовано для вывода на дисплей с помощью конкретного спрайта. Мультипликация достигается либо с помощью периодического изменения номера изображения для спрайта, либо изменением его местоположения. Различные части вашей программы могут изменять различные атрибуты спрайта, не затрагивая других атрибутов. См. приведенный ниже пример.

Пример:

```
100 GOSUB 1000' считать в A$ 5 разных изображений (1:5)
110' изображения — это пять различных положений бегущего человека
120 ON INTERNAL = 10 GOSUB 500
130 FOR X=1 to 5: SPRITE$(X) = A$(X) : NEXT
140' поместить спрайт
150 PUT SPRITE 10, (50, 50), 3, 1
160' переместить его
170 INTERVAL ON
180 FOR X=50 то 100
190 FOR Z=1 то 10: NEXT' убиваем время (задержка)
200 PUT SPRITE 10, (X, 50): NEXT :INTERVAL OFF
210 PUT SPRITE 10,,14' сделаем спрайт серым . . . остальная часть
    программы
500 SN = (SN+1) MOD 5
510 PUT SPRITE 10,,, SN+1: RETURN
```

См. также: SPRITE$ ( ), COLOR, CIRCLE, LINE, ON SPRITE GOSUB, Глава 4 по программированию графики.

Цель: Ввод данных, находящихся в тексте программы в операторе DATA

Формат: READ *перем.* [, *перем.*] . . .

перем :: = допустимое имя переменной (включая элемент массива).

Действие: Этот оператор читает данные, заданные оператором DATA в очередную переменную. При чтении алфавитно-цифровых данных в арифметическую переменную, вводится сообщение "Syntax error", "синтаксическая ошибка" и выдается номер строки с оператором READ, где обнаружена алфавитно-цифровая информация. В противном случае все данные, заданные в операторе DATA вплоть до запятой, считываются, переводятся во внутренний формат и присваиваются очередной переменной заданной в операторе READ. Этот процесс повторяется до исчерпания всех переменных списка в операторе DATA или до исчерпания данных. В последнем случае выводится сообщение об ошибке "Out of DATA", "данные исчерпаны".

Использование: Этот оператор является единственным методом доступа к данным, заданным в программе с помощью оператора DATA (см. DATA). Оператор READ просматривает данные, заданные оператором DATA за один проход от начала текста программы к концу. Порядок обнаружения операторов DATA может быть изменен с помощью оператора RESTORE, этот оператор позволяет вновь прочесть тот же самый элемент данных.

Пример:

```
100 DATA 34
110 READ A, B, C$, D$, E
120 DATA 234e-7 , goldfish
130 DATA "Testing, 1 . . 2 . . 3!"
140 PRINT A, B, C$, D$, E
1000 DATA-22
```

См. также: DATA, RESTORE

REM

оператор

Цель: Поместить комментарии в текст программы.

Формат: REM *текст комментария*

Действие: Этот оператор является оператором комментария в MSX-Бейсик. После ключевого слова REM может следовать любой текст, вплоть до конца данной строки. Этот оператор должен быть единственным в строке. Бейсик не производит абсолютно никаких действий при обнаружении REM: остаток строки пропускается. Номер строки с оператором REM доступен для использования в вашей программе. Иным способом задания комментария является апостроф (') . Хотя он и занимает один дополнительный байт по сравнению с комментарием, заданным оператором REM, он может быть размещен в любом месте, поскольку (') автоматически заканчивает предыдущий оператор, как если бы там находилось (:) .

Использование: После завершения написания и отладки программы вы можете создать вариант программы только для прогона, выпустив все комментарии. Это улучшит время выполнения программы и существенно уменьшит объем используемой памяти.

Пример:

```
10 REM SUPERPROG by S.E & GGL
20 REM This program does everything!
30 REM Method of operation:
....
250 SPRITE$(X) = A$ ' Setup image of walking man
```

См. также: Глава 4.

команда

RENUM

Цель: Перенумеровать строки в вашей программе.

Формат: RENUM [*новый*[[, *старый*] [, *шаг*]

новый :: = новый номер первой строки, подлежащей перенумерации
старый :: = старый номер первой строки, подлежащий перенумерации
шаг :: = шаг новой нумерации строк.

По умолчанию новый номер и шаг полагаются равными 10, а старый равным номеру первой строки в программе. Все три параметра являются номерами строк и принимают значения в диапазоне 0 = < номер строки < = 65529.

Замечание: Любой из приведенных параметров может быть опущен, но запятая, относящаяся к нему, должна быть сохранена. В качестве параметров "новый" и "старый" может быть использован символ (.). Это является ссылкой на текущий номер строки.

Действие: Эта программа используется для перенумерации всех строк в программе. Оператор RENUM кроме того, проверяет, все ли номера строк, на которые имелись ссылки в операторах Бейсика, действительно существуют (независимо от этого осуществляется перенумерация всех строк); при обнаружении отсутствия выдается сообщение "Undefined line nnnn in mmmm", "Отсутствует строка nnnn в mmmm"

Оператор RENUM начинает просмотр программы со строки с номером "старый" и перенумеровывает каждую строку с заданным шагом, выбирая в качестве начального номера "новый". Этот процесс продолжается до конца программы. Если "новый" меньше "старого", а также меньше других, не подлежащих перенумерации строк, выводится сообщение "Illegal function

call", "неверный вызов функции". Если ошибка не обнаружена, все строки в программе, начиная с номера, перенумеровываются.

Использование: Эта команда является хорошим средством управления текстом программы. Перенумеровать операторы можно для аккуратности, для слияния, и просто для того, чтобы проверить, что у вас есть и чего нет.

При перенумерации всей программы следует помнить, что обратного сделать нельзя. Если, скажем, подпрограммы обработки по ключу имели номера строк, легкие для запаминания (такие как 10000 с шагом 1000), тогда вы перестанете в них ориентироваться. Для защиты от этого вставьте номер строки перед нужной вам в качестве комментария с помощью оператора REM. Тогда вы легко найдете эти строки после перенумерации с помощью оператора LIST.

Если вы хотите перенумеровать всю программу, начните сначала и действуйте следующим образом. После выполнения каждого оператора RENUM распечатайте и найдите начало программы после перенумерованной части, снова используйте оператор RENUM, взяв в качестве начала параметр "старый". Таким образом, вы можете поместить программы обработки по ключу обратно точно в строки с номерами после выполнения оператора RENUM.

Вы можете использовать оператор RENUM с параметром "старый", превышающим все существующие номера строк, например, RENUM 65000, 65000. Перенумерация при этом не осуществляется, но вы получите информацию обо всех пропущенных номерах строк.

Цель: Сбросить указатель DATA для повторного чтения оператором READ

Формат: RESTORE [*номер строки*]

номер строки :: = любой допустимый номер строки 0—65529

Действие: Этот оператор изменяет состояние указателя, который сообщает оператору READ, какой из операторов DATA должен обрабатываться следующий раз. Оператор READ перемещает указатель вниз по мере ввода данных, пока не будут исчерпаны все операторы DATA. Оператор RESTORE возвращает указатель в начало, после чего все данные могут быть снова прочитаны оператором READ. Кроме того, в операторе RESTORE может быть задан номер строки, в этом случае указатель устанавливается в место, соответствующее строке с указанным номером. Оператор RESTORE может быть повторно использован в любой момент. Слово DATPTR (F6C8) содержит текущий указатель.

Использование: Прежде всего, можно восстановить начальное состояние указателя DATA для повторного чтения данных. По мере распределения пространства памяти в программе под сохранение данных, вы можете обращаться с ними как с массивом или устройством ввода, с которого можно вводить повторно; не следует использавать оператор READ для ввода данных в массив в вашей программе, поскольку занятое место удвоится. Нужно просто использовать данные повторно.

```
200 RESTORE : FOR Z=1 TO XX: READ ZZ: NEXT : RETURN
```

— читаем злемент данных *xx* в переменную *zz* без использования дополнительного пространства памяти в вашей программе.

Иной способ использования оператора RESTORE состоит в том, чтобы предоставить возможность каждой программе иметь свои собственные "локальные" переменные. Необходимо просто выполнять оператор RESTORE для установки указателя на начало данных подпрограммы при каждом обращении к ней. Однако, вы можете пожелать защитить указатель DATPTR, тогда переместите его перед выходом из подпрограммы, чтобы не изменять области данных других подпрограмм или основной программы, которые используют операторы DATA.

И, наконец, вы можете использовать оператор RESTORE как общий метод в программах обработки ошибок. В этом случае RESTORE позволяет указать на сообщение об ошибке — особенно, если данные представлены чем-нибудь типа музыки, длительно исполняемой в фоновой задаче.

Пример:

```
100 ON ERROR GOTO 500
. . .
500 IF ERR = 4 THEN RESTORE: RESUME 0
. . . .
1000 'first subroutine
1010 D1=PEEK (&HF6C8) : D2=PEEK (&HF6C9)
1020 RESTORE 1000
. . . .
1099 POKE & HF6C8, D1: POKE & HF6C9, D2
1100 RETURN' end of 1st.
```

См. также: DATA, READ

Цель: Выход из программы обработки ошибок

Формат:

```
                0-ноль
RESUME  [{ NEXT      }]
          номер строки
```

номер строки :: = любой существующий номер строки

0 :: = повторно выполняемый оператор, вызвавший ошибку;

NEXT :: = оператор, следующий за тем, который вызвал ошибку.

Действие: Этот оператор используется для возврата из программы обработки ошибок, заданной оператором ON ERROR GOTO. В каждый момент времени выполняется только один сегмент вашей программы (поэтому программа обработки ошибок совершенно не занимает длительной памяти при безошибочном выполнении программы). После того, как ваша программа обработки ошибок "решит" что делать, возможны следующие варианты:

RESUME или RESUME 0.
Повторяет действие оператора, выполнение которого ранее привело к появлению ошибки.
Предупреждение: Если вы не обнаружили причины появления ошибки, это приведет к зацикливанию, если только вы не сделаете следующего: запомните номер строки, где произошла ошибка (он хранится в переменной ERI). Потом выполните. Если снова произойдет ошибка, то потом перед выполнениемы проверьте, не произошла ли она в той же строке; если да— то откажитесь от обработки ошибки:

```
IF ERR = XX AND ERL = XY Then ON ERROR GOTO 0: RESUME
ELSE XX = ERR: XY = ERL: RESUME
```

RESUME NEXT
Продолжает выполнение вашей программы, начиная с оператора следующего за оператором, вызвавшим ошибку. Этот оператор может находиться в той же строке, где была обнаружена ошибка.

RESUME *номер строки.*
Вы можете продолжать выполнение вашей программы с любого удобного вам номера строки.

Использование: Вы должны использовать оператор RESUME для выхода из программы обработки ошибки и возврата в основную программу для обнаружения дальнейших ошибок. Прерывание из-за дальнейших ошибок в программе, запущенной по ON ERROR GOTO , невозможно до тех пор, пока не будет выполнен оператор RESUME.

Программы обнаружения ошибок могут также быть полезными для установки факта наличия или отсутствия файлов на диске.

Пример:

```
10 On ERROR GOTO 600
20 X = 3/Z ' Z is a new variable — this is a divide-by-zero
...
500 IF ERR = 11 AND ERL = 20 THEN Z = 1e-50 : RESUME
510 IF ERR < 15 THEN RESUME NEXT
520 ON ERROR GOTO 0: RESUME ' causes the error message
```

См. также: ERR, ERL, ERROR, ON ERROR GOTO

Цель: возврат из подпрограммы.

Формат: RETURN [*номер строки*]

номер строки :: = любой существующий номер строки в вашей программе.

Действие: Этот оператор передает управление оператору, следующему за оператором, вызвавшим эту подпрограмму, или за оператором, повлекшим такой вызов. В момент выполнения оператора возврата лобой незаконченный FOR-NEXT цикл в подпрограмме заканчивается и все их стековое пространство восстанавливается в начальное состояние. Оператор RETURN восстанавливает также стековое пространство, использованное оператором GOSUB. Подпрограммы могут иметь любой уровень вложенности и любую сложность.

Если данная подпрограмма является подпрограммой обработки прерываний, оператор RETURN вернет управление на условии, вызвавшее прерывание. Например, если данная подпрограмма была вызвана из-за столкновения спрайтов, то оператор RETURN передает управление обратно в главную программу и выполнит оператор SPRITE ON. Ваша программа обработки прерываний может вызывать другие подпрограммы; опреатор RETURN при этом возвращает управление в главную программу в нужный момент времени после обработки всех вызванных подпрограмм.

Иной формой оператора является RETURN "номер строки". Этот оператор возвращает управление из подпрограммы в точку вызова, как описано выше, и непосредственно после этого выполняет оператор GOTO "номер строки". Этот процесс можно представлять как извлечение из текста GOSUB -информации с

последущим выполнением GOTO (при этом следует помнить о том, что операторы FOR-NEXT, а также все средства обработки прерываний выполняются, как рассказано выше).

Использование: Каждая подпрограмма независимо от своего местоположения требует выполнение оператора RETURN для возврата в главную (вызывающую) программу. Если не сделать этого, то постепенно будет исчерпано все пространство памяти. Вы можете поместить в программе столько операторов RETURN, сколько необходимо, каждый из них может быть выполнен; наличие оператора RETURN в данном месте не означает физического конца подпрограммы. Однако, фактически для каждой подпрограммы должна быть единственная точка входа и единственная точка выхода по оператору RETURN; вся подпрограмма должна находиться между ними. Это приводит к более легкой отладке программы.

Пример:

```
100  ON SPRITE GOSUB 500: SPRITE ON
200  GOSUB 700: PRINT "We RETURNed to here"
 :
 .

500  'прерывание при SPRITE
510  RETURN ' возврат после прерывания по SPRITE

700  'Подпрограмма номер 1
710  IF X > 32 THEN RETURN 100 ' повторная защита
     программы.
720  RETURN 'возврат из подпрограммы
```

См. также: GOSUB, ON vat GOSUB, ON event GOSUB.

Цель: Возвращение правых символов строки.

Формат: *x$* = RIGHT$ (*стрвыр*, *арвыр*)

стрвыр :: = любое строковое выращение, строка-источник

арвыр :: = любое арифметическое выражение в диапазоне 0-255, задающее число возвращаемых символов.

Действие: Эта функция возвращает в точку вызова из строки, заданной параметром *стрвыр*, столько символов, сколько задано параметром арвыр, начиная справа. Порядок символов сохраняется.

Если *стрвыр* равно нулю, возвращается пустая строка. Очевидно, что RIGHT$ ( ) не может вернуть больше символов, чем содержала строка-источник. Эта функция начинает отсчет справа и не изменяет порядок символов при создании из них возвращаемой подстроки (зта подстрока занимает место в памяти, отведенной для строковых переменных). Если арвыр выходит за допустимые пределы, то выводится сообщение "Illigal function call" (неверный вызов функции).

Использование: Эта функция может быть заменена функциями LEN( ) и MID$ ( ), однако использование функции RIGHT$ экономит ваши силы. Например, для выделения младшей значащей цифры матиссы любого числа слева от десятичной точки можно задать;

```
X = VAL(RIGHT$(STR$(INT(num)),1))
```

Есть много аналогичных случаев, где удобно использование RIGHT$ ( ). Кроме того, функция RIGHT$ ( ) может быть полезна во многих операциях, ориентированных строго на

работу со строками: например, если у вас имеется 60-ти символьная строка, в которой последние 20 символов содержат фамилию.

Пример:

```
PRINT RIGHT$ (''abcd goldfish'', 8) goldfish
OK
```

См. также: LEFT$ ( ), MIDS ( ), LEN ( ).

Цель: Возвращает случайное число в диапозоне от 0 до 1

Формат: *x* = RND (*арвыр*)

арвыр :: = любое арифметическое выражение

Действие: Функция возвращает псевдослучайное число. Псевдозначение не является полностью случайным, однако, период повторения так велик, что его можно рассматривать как случайное.

Эта функция предоставляет отличное средство для игровых программ. Для параметра *арвыр* возможно три варианта:

арвыр < 0
Генератор случайных чисел запускается заново с новым исходным значением.
Для любого отрицательного аргумента осуществляется одинаковая инициализация генератора случайных чисел. Последующие обращения к генератору случайных чисел с помощью функции RND будут возвращать идентичный список случайных чисел.

арвыр > 0
Возвращается следующее случайное число из последовательности, заданной последней инициализацией генератора.

арвыр = 0
Повторяется вывод предыдущего случайного числа. Это удобно, поскольку таким образом можно сохранить последнее случайное число.

Все генерируемые случайные числа содержат 14 цифр и значения этих чисел заключены между 0 и 1 (это означает, что десятичная точка расположена слева от всех цифр).

Использование: Эта функция является основной для многих игровых и моделирующих программ. Для генерации случайного числа в диапазоне между x и y—1 надо задать Z = INT ((y—x) *RND (1)+x) Для получения целого между 0 и 9 надо задать Z = INT (10 *RND (1)).

Если для целей отладки вы хотите перед каждым прогоном получать одно и то же случайное число, поместите в начале этой программы строку вида Z = RND (*любое отрицательное число*).

Например: X = RND (—99)

Обычно желательно получать абсолюно непредсказуемые случайные числа. Чтобы добиться этого прежде всего необходимо инициализировать генератор случайных чисел в программе также случайным числом. Это можно сделать следующим образом:

X = RND (—TIME)

Помните, что такая инициализация должна осуществляться только один раз. Для дальнейшего получения случайных чисел используйте (1).

Пример:

```
10 X = RND (—TIME): INPUT "# of loops"; N
20 FOR X = 1 TO N: X1 = RND (1): X2 = RND (1)
30 IF X 1^2 + X 2^2 < 1 THEN IN = IN + 1
40 NEXT : PRINT "PI approx. = "; 4* IN/N
50 'Метод Монте Карло
```

См. также:

Цель: Запуск программы

Формат: RUN [{ *номер строки* }]
устрфайлимя

номер строки :: = любой реально существующий номер строки.
устрфайлимя :: = строковое выражение, задающее реально существующее устройство, и/или имя файла.

Действие: Это очень важная команда, введенная в режиме прямого выполнения, она переводит вашу программу в режим выполнения и начинает это выполнение с первой строки программы, выполняя сначала тор CLEAR для полной инициализации. Если эта команда задана как строка вашей программы, она перезапускает вашу программу таким образом, как если бы она была остановлена и запущена снова.

Если задан параметр номер строки, то выполнение программы начинается со строки с указанным номером. Если вы хотите начать выполнение программы с заданного номера строки, но при этом сохранить все переменные, не следует использовать эту команду — используйте для этого GOTO или GOSUB.

Если задана строковая переменная или константа, она интерпритируется как устройство и/или имя файла. Для детализации смотри BLOAD, BSAVE и LOAD.
По умолчанию в качестве устройства предполагается кассета. MSX Бейсик пытается загрузить программу, хранящуюся в формате ASCII с входного устройства; текущее содержимое памяти теряется, заданная программа грузится и выполняется идентичная команде этой команде за исключением того, что все открытые файлы так и остаются открытыми.

Использование: Эта команда используется всякий раз, когда нужно запустить программу. Ее отличительной особенностью является выполнение операции CLEAR, в результате чего осуществляется стирание значений всех переменных. Таким образом достигается идентичное состояние памяти при любом запуске программы.

Как замечено выше, эта команда может быть одним из предложений программы, это позволяет использовать ее для сцепления программных модулей и позволяет запускать различные фрагменты программы.

Пример:

```
RUN
Или
RUN 100: REM начать выполнение со строки с номером 100
```

См. также: BLOAD, BSAVE, LOAD, CLOAD, MERGE.

Цель: Вывоз программы в ASCII -формате.

Формат: SAVE *устрфайлимя*

устрфайлимя :: = любая допустимая строковая константа или выражение, задающее имя устройства и/или имя файла, где надлежит сохранить программу.

Действие: Так же, как и CSAVE, эта команда сохраняет текст вашей программы. Различие состоит в том, что текст может быть записан на любое существующее устройство; запись происходит в формате ASCII, после чего для нее могут быть использованы RUN “ ”, MERGE или LOAD.

ASCII -формат— это формат, в котором каждая строка представлена печатными символами в коде ASCII и заканчивается символами перевода строки и возврата каретки — (0 D), (0A). В конце программы добавляется символ конца файла (1A).

Для детализации смотри BLOAD, BSAVE, CLOAD и LOAD.

Использование: Основным способом использования этого предложения является сохранение программы в удобном для вас виде. Впоследствии ее можно слить с любой программой с помощью оператора MERGE, или вызвать на выполнение из другой программы с помощью LOAD, или RUN.

Дисковый Бейсик использует предложение в качестве основного метода сохранения программ во внутреннем формате (ASCII -формат с добавлением вслед “,A”).

Пример:

```
SAVE "CAS: name"
```

См. также: BSAVE, BLOAD, CSAVE, CLOAD, LOAD, MERGE, RUN.

Цель: Установить текущий режим экрана и другие параметры ввода/вывода.

Формат: SCREEN [, *режим*] [, *размер спрайта*] [, *звук клавиши*] [, *скорость ленты*] [, *печать*]

режим :: = арифметическое выражение со значением от 0 до 3
размер спрайта :: = арифметическое выражение со значением от 0—3
звук клавиши :: = арифметическое выражение со значением от 0—255
скорость ленты :: = арифметическое выражение со значением от 1—2
параметры печати :: = арифметическое выражение со значением от 0—255

Замечание: любой из параметров может отсутствовать, но запятая, относящаяся к нему, должна быть сохранена.

Действие: Этот оператор устанавливает текущий экран и некоторые важные параметры ввода/вывода. Если какой-либо из параметров выходит за допустимые пределы, выводится сообщение "Illegal function call" (неверный вызов функции); в противном случае устанавливается соответствующий параметр. Если параметр опущен, то он не изменяется; принимается старое значение.

Режим
Устанавливает режим работы экрана. Если параметр задан, то происходит очистка экрана и производится сброс всех регистров. На основе установки, заданной функцией BASE ( ), может также произойти стирание спрайтов. Существует 4 режима работы экрана:

SCREEN 0: только текст, граница отсутствует, максимум 40×24 символа, спрайты недопустимы, одноцветный.

SCREEN 1: поумолчанию; максимум 32x24 символа, спрайты допустимы, каждые 8 символов в наборе могут иметь цвет изображения/фона.

SCREEN 2: графический режим. Выводимый текст рисуется, 256x192 пикселей, каждые 8 пикселей имеют свой собственный цвет изображения/фона, спрайты допустимы, наиболее высокое разрешение.

SCREEN 3: графический режим, выводимый текст рисуется, 64x48 пикселей, каждый пиксель имеет свой собственный цвет, спрайты допустимы.

Только при работе экрана в режиме вывода текста — 0 и 1 допустимо применение операторов PRINT и INPUT. Только в графическом режиме — 2 и 3 экран воспринимает графические команды и операторы; допустимо также рисование текста с помощью специального устройства вывода GRP:. (Следует также иметь в виду, что оба графических экрана и все операции со спрайтами используют одну и ту же координатную сетку 0:255, 0:191). Следует отметить, что только SCREEN 0 поддерживает вывод символов размером 6x8 пикселей — экраны 1, 2 и 3 используют размер 8x8. Только экран 0 не допускает спрайтов, цветных границ или какого-либо другого индивидуального цветового управления (экран 1 имеет цветовое управление для группы символов, для чего необходимо использовать VPOKE). Однако, если это указание опущено, экран остается в текущем режиме. Для режимов SCREEN 0 и 1 можно использовать оператор WIDTH; после задания ширины она становится идентичной для обоих вариантов экрана.
Размер спрайта SCREEN 0 не допускает спрайтов. Для остальных режимов оператора SCREEN:

0: все спрайты 8x8 пикселей каждый использует 8 байт данных, допустимо 256 спрайтов
1: тоже самое, что и 0 за исключением того, спрайты выводятся на экран вдвое большего размера
2: все спрайты размером 16x16 пикселей каждый 32 байта данных; допусимто 64 спрайта
3: аналогично 2:, за исключением того, что спрайты выводятся вдвое большего размера.

Все спрайты должны быть одного и того же размера по числу байт и увеличению. Эти значения определяют размеры выборки из таблицы, находящейся в VRAM при задании псевдопеременной SPRITE$ ( ).

Звук клавишы
Если это значение было установлено в 0, то нажатие клавиш будет происходить без звукового сопровождения; любое другое значение (например 1) вызовет звуковое сопровождение нажатия клавиш (по умолчанию).

Скорость ленты
Это определяет скорость вывода (в бодах). Если задана 1, то устанавливается наиболее надежная скорость передачи — 1200 бод (по умолчанию). Если 2, то скорость становится 2400 бод — в два раза быстрее. Вы можете изменить скорость заданием команды CSAVE при вводе. Скорость определяется автоматически.

Печать
Этот параметр устанавливает тип печатающего устройства. Если он равен 0, то это определяет совместимость со стандартом MSX и принтер

будет распечатывать каждый символ из набора MSX. Если параметр не равен 0 (по умолчанию), то это несовместимое устройство, для простого печатающего устройства графические символы преобразуются в пробелы.

При обращении к оператору SCREEN могут быть заданы все параметры или любая комбинация; если параметры опущены (заменены запятыми) старые значения остаются неизмененными.

Использование: Этот оператор имеет много удобных возможностей — он может определить беззвучную клавиатуру, ускорить вывод на магнитный носитель (а затем и ввод с него), обеспечить преобразование графического вывода в текст в коде ASCII. Но основное его назначение — это определение типа экрана и размера спрайта. Вы можете изменить размер спрайта в любой момент — это не сотрет экран, но это сотрет определение спрайта! Точно так же вы можете использовать оператор SCREEN для выбора типа экрана и это не повлияет на спрайты, но это сотрет текущее содержимое экрана. (Следует отметить, что это так же прекратит и вывоз спрайтов, т.е. оператор PUT SPRITE нужно будет выполнить повторно).

Следует отметить, что оператор BASE ( ) позволяет определить размещение данных, выводимых на экран, в видеопамяти. В дальнейшем любое выполнение оператора SCREEN использует эти данные для вывоза на экран. Для понимания деталей см. описание BASE. Ширина текстовых экранов определяется командой WIDTH.

Пример:

```
100 MAXFILES = 2: SCREEN 2:
    PRESET (115, 92)
110 OPEN "GRP:" "AS #1:
    PRINT #1, "Заголовок" ' середина экрана
120 SCREEN 2: GOSUB 1000'
    определить спрайты, 16x16 увел. 1x
130 CIRCLE (50, 50),
    25: PAINT STEP (0, 0)
140' спрайты и графика хорошо выводятся на экране 2
150 SCREEN,,,2:  OPEN "CAS: дан"
    FOR OUTPUT AS #2 'быстрый
```

См. также: WIDTH, BASE ( ), VDP ( ), PUT SPRITE, SPRITE$( ), Главу 4 описание графического режима.

Цель: Возвращает значение —1,0 или 1 в зависимости от знака аргумента

Формат: *x* = SGN (*чис. выр*)

чис. выр :: = любое числовое выражение

Действие: Эту функцию называют "сигнатурой". она выполняется следующим образам:

чис. выр < 0
возвращает -1

чис. выр = 0
возвращает 0

чис. выр > 0
возвращает +1

Использование: В основном эта функция применяется при " множении" одного числа на знак другого числа. Она позволяет также приравнять нулю реэультат, если один из операндов равен нулю (без использования оператора IF), см. пример).

Пример:

```
100 INPUT A
110 PRINT 35*B*A*SGN(A)
```

(для тех случаев, когда A—отрицательное число, а вы не хотите, чтовы B изменило знак).

Цель: Возвращает синус угла, заданного в радианах.

Фармат: *x* = SIN (*арифмвыр*)

арифмвыр :: = любое арифметическое выражение.

Действие: Эта функция преобразовывает аргумент и рассматривает его как угол в радианах. Используя это значение, она возвращает синус угла. Эта функция дает двойную точность и, таким образом, весьма точная.

Использование: Эта функция является полезней при работе с углами и длинами. Преобразования углов в радианы производится по следующему правилу:

X*PI/180, где PI = ATN (1) * 4 или 3.1415926535898

Смотрите приложение для производных функций и краткое объяснение того, где используются тригонометрические возможности.

Пример:

```
PRINT SIN (PI/2)
1
OK
```

См. также: COS ( ), TAN ( ), ATN ( ), приложение по тригонометрическим функциям.

Цель: Обеспечивает прямой доступ к звуковому генератору.

Формат: SOUND *регистр*, *установка*

регистр :: = арифметическое выражение, дающее регистр, 0—13
установка :: = арифметическое выражение для нового значения, 0—255

Действие: Эта команда позволяет квалифицированному пользователю управлять Программируемым Звуковым Генератором (PSG) совместимым с General Instrument AY-3-8910.
Имеется четырнадцать 8-битовых регистра, пронумерованных от 0 до 13, каждый из которых предназначен только для записи и может содержать значения в диапазоне от 0 до 255.
При детальном понимании любой звук может быть сгенерирован, включая широкое разнообразие эффектов; все указанное является лишь кратким введением, а не инструкцией по звуковоспроизведению.

Ниже приводится сводка регистров:

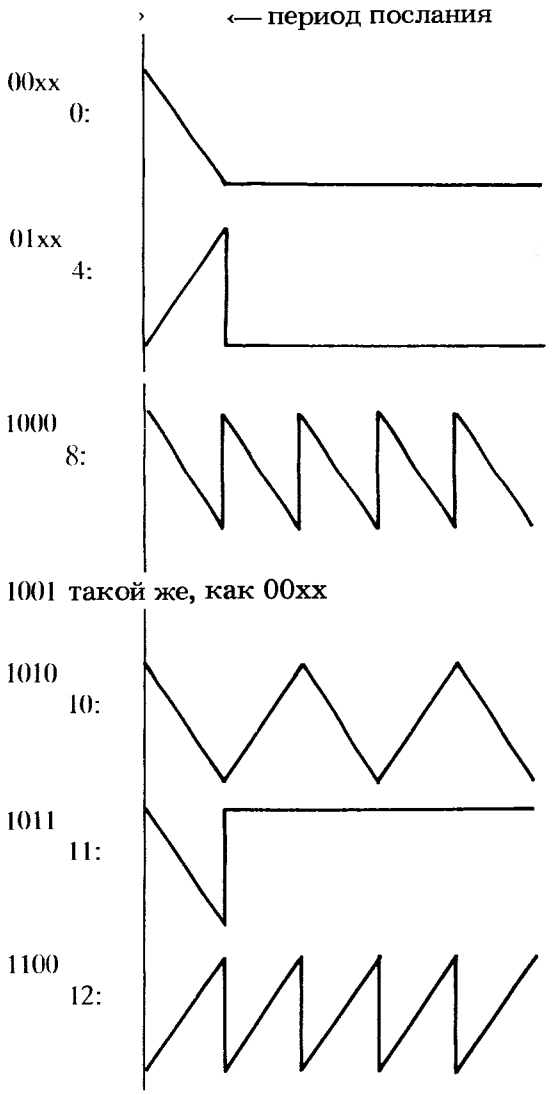
0: нижний 8 бит частоты голоса A
1: верхние 4 бита частоты голоса A (старшие 4 бита не используются)
2: нижний 8 бит частоты голоса B
3: верхние 4 бита частоты голоса B (старшие 4 бита не используются)
4: нижние 8 бит частоты голоса C
5: верхние 4 бита частоты голоса C (старшие 4 бита не используются)
6: пятибитовый Период Шума (NP) для генератора шума (стршие 3 бита не используются)
7: управляющий регистр смешивания: старшие 2 бита всегда устанавливаются 10 двоичное. Младшие 3 бита обеспечивают возможность управления тоном; средние три бита обеспечивают возможность управления шумом. Порядок голосов — CBA (голос A имеет наименьшее значение). Наличие нуля разрешает смешивание; единица запрещает звук-голос.
8: управление амплитудой и режимом канала A; нижние 5 бит выглядят MAAAA — AAAA — это четырехбитовая амплитуда (взвешенная), M — бит режима: если 0, то выбирается фиксированная амплитуда из AAAA; если 1, то выбирается управление переменной амплитудой в соответствии с выбором конверта. Или предположения, что M=0 открывает AAAA; M=1 открывает управление от конверта.
9: управление амплитудой и режимом канала B. Также как и выше.
10: Управление амплитудой и режимом канала C. Также как и выше.
11: Младшие 8 бит управления периодом конверта (частота, EP).
12: Старшие 8 бит управления периодом конверта (частота, EP).
13: Младшие 4 бита управляют формой конверта. См. диаграмму.

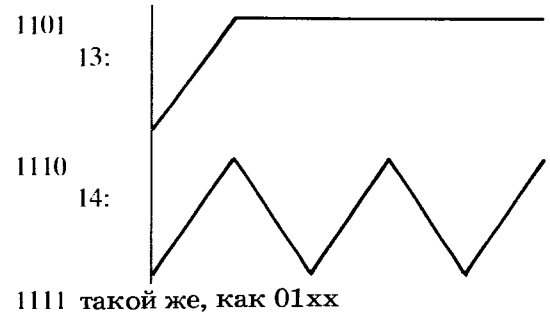
Ниже приводятся правила расчета музыкальной частоты, которую можно получить при любой 12-ти битной установки: музыкальная частота = 124797/частота тона, где частота тона есть, скажем, 256 рег.: 1+рег. 0.

Аналогичные вычисления с использованием периода шума вместо частоты тона дадут вам частоту инерируемого шума. Аналогично, частота генерируемого послания может быть вычислена по формуле: частота послания в Гц=7799.8/период послания, где период послания (EP) определяется по формуле 256*рег. 12+рег.11.

Ниже приведены восемь допустимых форм волн:

рег. 13 (двоичное, X=не имеет значения)
Форма послания





1111 такой же, как 01xx

Использование: Этот оператор представляет возможность программ, написанным на MSX Бейсик воспроизводить звуки, дополнительные к нормальным типам, которые воспроизводятся с помощью оператора PLAY. Однако, при использовании этого оператора, временные характеристики явл. критичными, не менее важным является понимание побочных эффектов при выдаче звуковых сигналов. Несколько примеров, приведенных далее, объясняет вам указанную мысль. Всегда необходимо помнить, что BEEP (STOP) и PLAY изменяют звуковые регистры.

Пример:

```
50 FOR X=0 TO 13: SOUND X, 0: NEXT' очистить все регистры.
100 SOUND 7, 62: SOUND 8, 15: SOUND 9, 16: SOUND 10, 16
110 SOUND 12, 16: FOR X=48 TO 170: A=5^5^5' 25 ms задержка
120 SOUND 0, X: NEXT: SOUND 8, 0' эффект падающей бомбы.
125'
130 SOUND 0, 0: SOUND 6, 15: SOUND 7, 7: SOUND 12, 16
140 FOR X=8 TO 10; SOUND x, 16: NEXT
150 SOUND 13,0: FOR X=1 TO 500: NEXT' эффект взрыва.
155'
160 SOUND 1, 1: SOUND 7, 46: SOUND 8, 15: SOUND 9, 9
170 FOR X=80 TO 33 STEP-1: POKE 0, X: NEXT
180 X=TAN (1): X=TAN (1) ' 2X 175 ms задержка
190 FOR X=55 TO 40 STEP-1
200 POKE 0, X : A=5^5: NEXT '12ms задержка
210 FOR X=40 TO 100: POKE 0, X: NEXT
220 SOUND 8, 0: SOUND 9, 0' эффект свиста.
230 END' вызывает STOP, далее BEEP и все звучание прекращается.
```

См. также: PLAY, BEEP

Цель: Создание строки пробелов

Формат: *x$* = SPACE$ (*длина*)

длина : : = арифметическое выражение в диапазоне 0—255

Действие: Эта функция создает строку указанной длины. Если длина равна нулю, то создается нулевая строка; если длина вне допустимого диапазона, то выводится сообщение об ошибке "Illegal function call" (неверный вызов функции). Вновь созданная строка помещается в область строк и заполняется пробелами — ASCII (20).

Использование: Аналогичный результат можно получить с помощью STRING$ (длина," "), но использование данной функции выглядит лучше. Единственным недостатком явл. то, что вы должны иметь достаточно дополнительного места в области строк для размещения строки, создаваемой по этой функции, в дополнении ко всему остальному. Используйте эту функцию каждый раз, когда вы хотите добавить поля пробелов.

Пример:

```
100 NAME$ = LEFT$ (NM$+SPACE$ (30), 30)
```

(выполняется юстировка влево и дополнение имени MM$ до фиксированной длины 30 байтов).

См. также: SPC ( ), STRING$( )

SPC

фукция

Цель: Используется только в операторе PRINT для печати пробелов.

Формат: PRINT . . . SPC (*длина*) . . .

длина :: = арифметическое выражение в диапазоне 0—255

Действие: Эта функция используется только в операторе PRINT. Она создает и печатает заданное длиной число пробелов. Она не создает новых строк и таким образом не требует дополнительного места для выполнения.

Использование: Эту функцию предпочтительно использовать с оператором PRINT, т.к. она не требует никакой дополнительной строковой области для работы. Одной из причин удобства использования этой функции является то, что оператор не всегда очищает экран от ранее выведенных символов или символов, уже имеющихся на экране. С помощью этой функции производится фактический вывод пробелов, слева от выводимого на экран элемента. Так SPC (5) идентично " ␣␣␣␣␣ " в операторе PRINT.

Пример:

```
220 ZZ = 8' установить промежуток между колонками
230 PRINT SPC (ZZ); AX; SPC (ZZ); BX; SPC (ZZ)
```

См. также: PRINT, SPACE$( ), STRING$ ( )

оператор

SPRITE ON / OFF / STOP

Цель: Управление процессом прерывания при столкновении спрайтов[1].

Формат: SPRITE $\begin{Bmatrix} \text{ON} \\ \text{OFF} \\ \text{STOP} \end{Bmatrix}$

Действие: Оператор SPRITE ON/OFF/STOP управляет процессом прерывания при столкновении спрайтов. Когда задано SPRITE ON, любое столкновение вызывает прерывание (по окончании выполнения текущего оператора) и затем переход к подпрограмме, определенной в ON SPRITE GOSUB. Автоматически устанавливается SPRITE STOP; когда происходит RETURN из обслуживающей подпрограммы, вновь устанавливается SPRITE ON до тех пор, пока вы явно не отключите обработку оператором SPRITE OFF.

SPRITE STOP временно отключает обработку прерываний. Если столкновение спрайтов произошло, то этот факт фиксируется, но никаких действий не происходит, однако, если позже вы выполните SPRITE ON, то будет обрабатываться ранее зафиксированное прерывание. Наиболее простым оператором является SPRITE OFF: после выполнения этого оператора MSX-Бейсик игнорирует любое столкновение спрайтов до тех пор, пока вы не начнете все сначала с заданием SPRITE ON.

При этом вы должны задать оператор ON SPRITE GOSUB, иначе это не даст никакого эффекта.

Использование: Этот оператор является частью системы обработки прерываний в MSX-Бейсике. Конечно, ничего не произойдет, пока указанный оператор не будет использован, так как оператор ON

[1] прим. перев. Спрайт (SPRITE)— динамический перемещающийся объект экрана

SPRITE не управляет системой прерываний, а запускает только данный оператор.

Столкновение спрайтов неизбежно происходит сразу же после задания PUT SPRITE. Очень трудно определить точно, какой спрайт вызвал столкновение, поэтому рекомендуется поступать следующим образом:

sn = 12:PUT SPRITE sn,...

Если вы определили номер спрайта с помощью стандартной переменной и затем использовали только эту переменную в PUT SPRITE, тогда ваша программа обработки прерываний при столкновении спрайтов может определить, кто вызвал это столкновение проверкой этой переменной. Если же вы хотите выяснить, кто пострадал, то потребуются сложные приемы, включающие сохранение массива размещений каждого спрайта или многочисленных VPEEK. Следует отметить, что столкновение спрайтов происходит только тогда, когда сталкиваются видимые части.

Пример:

```
100  ON SPRITE GOSUB 1000
110  SPRITE ON
200  SPRITE STOP ' принять без выполнения отработки
210  GOSUB 2000: SPRITE ON
250  SPRITE OFF  ' игнорировать столкновения
260  GOSUB 3000: SPRITE ON
...
1000 ' подпрограмма для обработки столкновений
1010 IF CND THEN SPRITE OFF: RETURN ' нет столкновения
1020 ' есть столкновение
1030 RETURN
2000 ' подпрограмма во время работы которой столкновения не
     обрабатываются
2010 ' однако, происходит контроль их наличия и отработка
     столкновения после возврата
3000 ' подпрограмма, которая сбрасывает все спрайты — без
     проверки
3010 ' на столкновения до своего окончания
```

Смотри также: ON событие GOSUB, SPRITE$ ( ), PUT SPRITE

Цель: Позволяет установить и проверить шаблон спрайта.

Формат:

SPRITE$ (*числ*) = *строчное выражение*
или
*x$* = SPRITE$ (*числ*)

числ :: = арифметическое выражение в диапазоне
в диапазоне
0—255 для SCREEN 0 или SCREEN 1
в диапазоне
0—63 для SCREEN 2 или SCREEN 3
строчное выражение :: = любое строчное выражение

Действие: Эта псевдопеременная может скорее рассматриваться как псевдомассив — она работает как массив строк. Она представляет вам доступ к шаблонам спрайтов — не к видимым спрайтам, но к данным находящимся в видео-памяти, которые определяют вид спрайтов. Допускается наличие 64-х 32-байтных спрайтов или 256 8-байтных спрайтов и SPRITE$ ( ) оперирует с 8-ю и 32-мя байтами одновременно. Число и размер спрайтов устанавливается оператором "SCREEN, размер спрайта" (который случайно может очистить все данные спрайта). Если задан размер 0 или 1, то спрайты—8-байтовые, а если он 2 или 3— 32-х байтовые. (Если вы подсчитаете, что указанные значения определяют 2-х килобайтный размер VRAM, используемый для этой псевдопеременной).

Если задано SPRITE$ (числ) =, то вычисляется строчное выражение и его байты пересылаются в соответствующий значению "числ" образ спрайта. Если строка слишком велика, то она укорачивается, а если коротка — то производится дополнение справа нулевыми байтами (00) до длины спрайта.

Если SPRITE$ (числ) используется в каком-либо выражении, то он возвращает 8 или 32 бата содержимого видео-памяти, соответствующей значению "числ" как любая другая строковая операция. Фактически SPRITE$ можно рассматривать точно также как массив строк фиксированной длины: DIM SPRITE$ (255) 8 или DIM SPRITE$ (64) 32 (последние цифры — это длины элементов в байтах).

Если спрайты имеют длину 32 байта, то данные организуются в две колонки: первые 16 байт представляюти левую половину спрайта, а последние 16 байт представляют правую половину спрайта, в каждой половине 8 байт (1 байт, 8 точек) ширина и 16 точек (байт) высота. Причем значение "1" бита — это видимая часть спрайта, а значение "0" — прозрачная часть.

Использование: Прежде всего это первичный способ помещения образов ваших спрайтов в видеопамять, предназначенную для хранения шаблонов спрайтов. Будьте внимательны и прежде всего установите размер вашего спрайта, чтобы не уничтожить все содержимое этой памяти. И, поскольку, задание 32-х байт в двоичном виде всегда затруднительно, используйте оператор ДАТА для чтения этих данных.

Другое возможное использование SPRITE$ — это качество строкового массива для программ с нехваткой памяти. Необходимо при этом помнить, что при возвращении данных из области спрайтов, при использовании выражения = SPRITE$ ( ), информация временно копируется в массив строки, несколько увеличивая таким образом размер этого массива.

См. также: PUT SPRITE, DATA, ON SPRITE GOSUB, SPRITE, Глава 4 секции графики.

Цель: Выдаёт квадратный корень ее аргумента

Формат: *x* = SQR (*арифм выр.*)

арифм выр :: = любое арифметич. выражение

Действие: Эта функция вычисляет квадратный корень заданного вами арифметич. выражения. Она дает не слишком высокую точность, но все же более высокую, чем X^.5 — мы уже говорили о 13-ой цифре которая является не слишком серьезной ошибкой.

Эта функция работает с двойной точностью. Конечно, MSX-Бейсик не распознает мнимые числа и, поэтому, попытка обращения к функции с отрицательным значением аргумента вызовет сообщение об ошибке "Illegal function call"[1].

Использование: Этой функцией нужно пользоваться, если вы хотите получить несколько более высокую точность или сделать программу более читаемой.

Пример:

```
100 INPUT"ax^2+bx+c=0 — задайте a,b,c";A,B,C
110 DISC=B*B-4*A*C:W=1/ (2*A): Z=-B/ (2*A)
120 IF DISC=0 THEN PRINT "Двойной корень, x= "; Z: GOTO 100
130 IF DISC < 0 THEN 160
140 PRINT "корни:"; Z+SQR (DISC)*W;
150 PRINT "и"; Z-SQR (DISC)*W: GOTO 100
160 PRINT "мнимые корни:"; Z;"+/-"; SQR (-DISC)*W; "i"
170 GOTO 100
```

См. также: Глава 2 о выражениях.

[1] прим. перев. Неверный вызов функции.

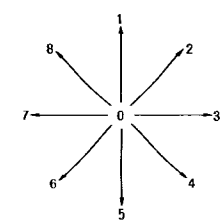
Цель: Выдаёт состояние джойстика.

Формат: *x* = STICK (*число*)

который :: = арифметическое выражение в диапазоне 0—3, 0 = клавиатура; 1 и 2 = разъёмы джойстика 1 и 2.

Действие: Если значение "число" находится вне допустимого диапазона, то выводится сообщение об ошибке "ILLEGAL FUNCTION CALL"[1]. В противном случае эта функция читает состояние джойстика и выдаёт число от 0 до 8, отмечающее перемещение рукоятки джойстика. Джойстики присоединяются к MSX компьютору через специализированные порты джойстика, типа ATARI/COMMODORE. Цифры опередеяют 8 направлений, по которым можно производить перемещение (в отличие от аналоговых джойстиков, которые допускают сотни различных позиций). MAX BASIC позволяет вам использовать четыре клавиши управления курсором клавиатуры как джойстик — STICK 0. (следует иметь в виду, что состояние кнопки джойстика считывается функцией STRIG ( )).

Следующий рисунок демонстрирует различные значения функции STICK.



[1] прим. перев. Неверный вызов функции.

Для клавиатуры (STICK (0)) клавиши вверх, вниз, влево и вправо выдают, соответственно, значения 1, 5, 3 и 7. Нажатие одновременно двух соседних клавиш дает значения 2, 4, 6 и 8. Задание функции STICK (1) и STICK (2) выдаёт состояние джойстиков, подключенным к портам 1 и 2. 0 означает отсутствие нажатия какой-либо клавиши.

Использование: Функция STICK — вместе с функцией STRIG — позволяет вам получить состояние периферийного устройства быстро и просто непосредственно из BASIC'а. Наиболее быстрый путь определения нажато ли что-либо:
IF STICK (1) THEN (нажато) ELSE (не нажато).

Пример:

```
100 REM программа для работы с клавишами управления курсором в
    качестве джойстика.
110 D=STICK (0)' чтение состояния клавишикурсора
120 IF D=0 THEN 110 ' ничего не нажато
130 IF D=8 OR D=1 OR D=2 THEN Y=-1
140 IF D=4 OR D=5 OR D=6 THEN Y=1 ' для клавиш перемещения
    курсора вверх и вниз.
150 IF D=6 OR D=7 OR D=8 THEN X=-1
160 IF D=2 OR D=3 OR D=4 THEN X=1 ' для клавиш перемещения
    курсора влево или вправо
170 PSET STEP (X, Y), C
```

См. также: STRIG ( ), PAD ( ), PDL ( )

Цель: Останов выполнения с выводом сообщения и звонка (BEEP)

Формат: STOP

Действие: Этот оператор останавливает выполнение и переводит программу в режим доступа с терминала. Он выводит сообщение BREAK IN zzzz, где zzzz — это номер строки, содержащей STOP. Он также выводит предупредительный звонок (BEEP). Если вы хотите продолжить выполнение вашей программы (т.е. если существует оставшаяся после STOP часть программы) вы можете воспользоваться командой CONT — ни в коем случае не изменяйте, не добавляйте и не исключайте строки программы!

Использование: Оператор STOP является очень полезным отладочным средством. Вы можете свободно размещать эти операторы в нужных точках вашей программы в любой момент, когда вы получаете сообщение BREAK, вы можете проверить, где вы находитесь (верно ли был выполнен переход? Там ли вы находитесь, где вы хотите быть и распечатать (PRINT) значение переменных с помощью непосредственно вводимых команд (те ли значения они имеют?) Когда вы закончили отладку, вы можете ввести CONT и продолжить выполнение после оператора STOP (при условии, что вы не изменяли ни одной программной строки). Нажатие клавиши Control-STOP выполняет те же действия, как и ввод оперетора STOP в вашу программу.

STOP

оператор

Пример:

```
100 ' линия 1
105 STOP
110 ' линия 2
115 STOP
120 ' линия 3
125 STOP
...
530 IF CND THEN STOP ELSE 100
```

Си. также: RUN, CONT

STOP ON / OFF / STOP

Цель: Управление процессом прерывания при вводе Control-STOP

Формат:

```
      ON
STOP {OFF}
      STOP
```

Действие: Этот оператор используется для управления процессом прерывания, вызываемым вводом Control-STOP. Он также обеспечивает вас механизмом защиты пользователя от прерывания программ во время выполнения. Естественно, что вы должны предварительно задать оператор ON STOP GOSUB, поскольку иначе данный оператор не даст никакого эффекта.

STOP ON запускает обработку прерывании. Тогда в любой момент, при вводе Control-STOP Бейсик прервет выполнение программы по окончании исполнения текущего оператора. А затем управление будет передано подпрограмме, заданной в операторе ON STOP GOSUB. Кроме того, будет автоматически выполнен оператор STOP STOP; по окончании выполнения подпрограммы неявно выполняется STOP ON до тех пор, пока STOP OFF не отключит его.

STOP STOP временно приостанавливает обработку прерываний, но поддерживает состояние контроля. Если клавиши нажаты, то этот факт фиксируется. И тогда после выполнения следующего STOP ON, производится обработка прерываний.

STOP OFF отключает всякую обработку прерываний. В этом случае BASIC игнорирует ввод Control-STOP как сигнал прерываний. Если в дальнейшем появляется STOP ON, то весь процесс продолжается, как это было описано выше.

Если после оператора ON STOP GOSUB стоит непосредственно оператор RETURN, то отключается возможность выполнения какого-либо прерывания со стороны пользователя. Однако не следует забывать о байте ENSTOP, находящемся по адресу (FBB0); если он не равен нулю, то допускаестя 4-х клавишное прерывание, никак иначе не реализуемое.

В процессе обработки прерываний от Control-STOP, работающий оператор INKEY$ иногда воспринимает Control-C (03); что же касается функции INPUT$ ( ), то она всегда воспринимает этот символ.

Использование: Обработка прерываний с помощью STOP может быть использована для нескольких целей. Во-первых, вы можете защитить пользователя от прерывания программы. Во-вторых, его можно использовать как общий метод завершения программы. И в-третьих, он позволяет вам читать вводимые клавиши с помощью INKEY$ ( ). Следует помнить, что данный оператор не влияет на работу клавиши STOP-функции пауза; что же касается INPUT$ ( ), то она отключает этот оператор (что является единственным способом его отключения).

Пример:

```
100 ON STOP GOSUB 110 : STOP ON: GOTO 120
110 RETURN
120 ' программа продолжается
210 A$=INPUT$ (1) ' будет прочитан Control-STOP kak (03)
```

См. также: ON событие GOSUB

Цель: Чтение состояния пусковых кнопок джойстиков.

Формат: *x* = STRIG (*числ.*)

числ. :: = арифметическое выражение в диапазоне от 0 до 4.

Действие: С помощью этой функции можно определить состояние пусковых кнопок ваших джойстиков. Может быть считано состояние 3-х джойстиков: "встроенного", если это клавиатура и двух дополнительных, которые могут быть подсоединены к портам 1 и 2. Значение "числ.", которое может быть 0, 1 или 2, определяет пусковую кнопку, состояние которой должно быть прочитано. Для клавиатуры, используемой в качестве "встроенного джойстика, пусковым устройством является клавиша "пробел".

Функция принимает значение либо ноль (FALSE), если кнопка не нажата, либо 1, если кнопка нажата. Следует отметить, что при этом не считывается местоположение джойстика, которое может быть определено с помощью функции STICK ( ). Любой конкретный MSX компьютер всегда имеет "встроенный" джойстик 0; пользователь всегда может подключить джойстик фирм ATARI/COMMODORE к потру 1, как дополнительное оборудование, однако MSX системы низкой стоимости могут не иметь порта 2 для подключения еще одного джойстика.

Если соответствующий джойстик подключен к портам 1 или 2 (это может быть также соответствующим образом распаяный джойстик фирм APPLE/IBM или специальный контроллер KONAMI), то вы можете прочесть состояние сразу двух пусковых кнопок. Задайте "числ" значение 3 или 4 для чтения второй кнопки джойстика, подсоединенного к портам 1 или 2.

Любое другое значение "числ" вызовет сообщение об ошибке "ILLEGALE FUNCTION CALL"[1].

Использование: STRIG ( ) также как и STICK ( ) позволяют вам считывать состояния внешних устройств непосредственно из BASIC'а без каких-либо специальных усилий, при этом могут считываться несколько различных типов устройств; все, что необходимо для этого, что эти устройства должны быть соответствующим образом распаяны.

Что касается пусковых устройств джойстиков, то имеется 3 возможности работ с ними: во-первых, использовать INKEY$ или какой-либо другой способ ввода для чтения клавиши "пробел", во-вторых, использование STRIG ( ) для определения факта включения какого-либо пускового устройства и в-третьих, использовать ON STRIG GOSUB для выполнения прерывания всякий раз, когда нажата кнопка. Разница между двумя последними способами заключается в том, что вы хотите опередилить: только факт запуска или этот же факт, но в нужном месте вашей программы.

Пример:

```
100 IF STRIG (1) THEN GOSUB 1000 ELSE 100 ' не готов
```

См. также: ON STRIG GOSUB, STRIG ON/OFF/STOP, STICK ( ), PDL ( ), PAD ( )

[1] прим. перев. Неверный вызов функции.

Цель: Управление прерываниями кнопки джойстика

Формат:

STRIG (*числ*) {ON / OFF / STOP}

числ :: = арифметическое выражение в диапазоне 0—4.

Действие: Этот оператор управляет обработкой прерываний от кнопок джойстиков. Предварительно должен быть задан оператор ON STRIG GOSUB, иначе данный оператор не дает никакого эффекта. С помощью этого оператора вы можете разрешать и запрещать процесс обработки прерываний при нажатии кнопки джойстика, в этом случае программа прерывается по окончании текущего оператора и происходит переход в подпрограмму, указанную в операторе ON . . .

Детальное рассмотрение вариантов работы оператора приведено в аналогичных описаниях SPRITE ON/OFF/STOP и KEY(n) ON/OFF/STOP.

Основное отличие заключается в том, что вы можете независимо задавать параметры ON/OFF/STOP для пяти различных прерываний, используя значение "числ" для указания номера кнопки, прерывания от которой вы хотите обрабатывать. Объяснение 5-ти различных значений параметра "числ" приводится в STRIG ( ).

Использование: Для данного оператора обработка прерываний понимается в том смысле, что вы можете менять отдельную подпрограмму, выполняющую то, что вы хотите в тот момент, когда нажимается кнопка джойстика. (Не забудьте, что пробельная клавиша может также являться пусковым устройством). Таким образом, вы не должны заботиться о том, когда проверить пусковое устройство — MSX BASIC сделает это автоматически и немедленно отреагирует.

STRIG ON / OFF / STOP

оператор

Пример:

```
100 ON STRIG GOSUB 1000, 2000, , 4000: STRIG (0) ON
110 STRIG (1) ON: STRIG (3) ON
...
1000 ' эта часть работает, когда нажата пробельная клавиша
1100 RETURN
2000 ' эта часть работает, когда нажата кнопка 1 джойстика 1.
2100 RETURN
4000 ' эта часть работает, когда нажата кнопка 2 джойстика 1.
```

См. также: ON событие GOSUB, STRIG ( ), STOP ON/OFF/STOP, KEY (n) ON/OFF/STOP

фукция

STR$

Цель: Преобразование цифровых данных в строчный формат ASCII

Формат: *x$* = STR$ (*арифм. выр*)

арифм. выр :: = любое арифметическое выражение

Действие: Этот оператор похож на отложенный оператор PRINT. "Арифм выр" обрабатывается как обычно. Затем данные "печатаются" в новую временную строку в области строк, что и является возвращаемым значением функции. (Не путайте ее с функцией STRING$ ( ), которая создает длинную инициализированную строку.

Форматирование функцией STR$ ( ) выполняется аналогично форматированию в PRINT, например, если N = 1/3, то STR$ (N) выдаст строку вида ".33333333333333" — и все эти ,,. тройки являются действительно ASCII (33) = 51 десятичное, т.е. символом, определенным как "3". Однако, есть единственное отличие между PRINT и этои функцией: она не выдает автоматически следующего пробела за числом.

Этот оператор выполняет действие противоположное оператору VAL. VAL (STR$ (n)) возвращает n, а STR$ (VAL (nm$)) возвращает nm$, если nm$ является строкой, содержащей допустимое число.

Использование: В MSX BASIC — задание вида PRINT n ; "X" создает пробел между последней цифрой n и X. Для исключения этого лишнего пробела можно воспользоваться оператором вида:

```
PRINT STR$ (n); "X"
```

Еще одним использованием данной функции является получение чисел в строчной форме, что

позволяет обрабатывать каждый символ (цифру) отдельно с помощью строчных функций. Вообще говоря, вы можете выполнить то же самое с помощью специальных числовых операций, используя INT ( ), но с помощью данной функции это сделать проще. Например, для получения двух младших цифр целого попробуйте воспользоваться выражением вида:

```
X=VAL (RIGHT$ (STR$ (IN), 2))
```

И, наконец, при сборке полных "записей" в строки такие, что каждый элемент в массиве строк содержит имя, адрес и некоторые числовые данные, функция STR$ ( ) позволяет наиболее быстрым путем добавить числовые данные в виде, понятном человеку (т.е. занимающим большее место, чем внутреннее предетавление). Кроме того, преобразование выполняется быстро и просто.

Пример:

```
100 PRINT STR$ (A); TAB (10); STR$(B); TAB (20)
110 ' выше указано, что числа, состоящие из 10 цифр будут размещены
      в колонки.
320 AB$(N)=NM$+AD$+STR$(SS)+STR$(ZP)
450 FOR X=1 TO MAX: PRINT #1, AB$(X): NEXT
```

См. также: VAL ( ), PRINT

Цель: Возвращает инициализированную строку любой длины.

Формат: *x$* = STRING (*длина*, $\left\{\begin{matrix}\textit{арифм выр}\\ \textit{стр выр}\end{matrix}\right\}$)

длина :: = арифметич. выражение в диапазоне 0—255, задающее длину создаваемой строки.

арифм выр :: = арифметич. выражение в диапазоне 0—255, задающее код ASCII символа заполнителя новой строки

стр выр :: = любое строчное выражение: 1-ый символ рассматривается как символ — заполнитель новой строки

Действие: Назначение функции STRING$ — это создание строки любой длины, в которой символ получает значение любого указанного кода. Все символы в новой сгенерированной строке идентичны. Генерируемый символ представляет собой любой допустимый код ASCII от 0 до 255; в качестве альтернативы вы можете указать действительный символ какой вы хотите — это особенно важно, если вы не помните его цифрового значения. Тип второго аргумента определяет способ, которым вы задаете символ-заполнитель.

Если вы хотите инициализировать новую строку пробелами, то три приведенных ниже способа выполняют идентичную операцию:

```
SPACE$ (41) = STRING$ (41, 32)
= STRING$ (41, "  ")
```

Во всех случаях, когда арифметические параметры находятся вне допустимых диапазонов, появляется сообщение об ошибках "ILLEGAL FUNCTION CALL". Если указана длина 0, то генерируется нулевая строка, независимо от того, какой символ заполнитель вы задали.

Использование: Имеет место рекурсивная задача в MSX Бейсике; которая сводится к следующему:

```
FOR x = 1 TO 100: a$ (x) = " .....": NEXT
```

Это выражение ничего не инициализирует. Все элементы строчного массива указывают на одну и ту же строку точек в тексте программы! Если вы хотите создать 100 строк в строчной области, воспользуйтесь следующим выражением:

```
FOR x = 1 TO 100: a$ (x) = STRING (10, " . "): NEXT
```

Вы должны также заметить, что два нижеследующих задания функции идентичны:

```
STRING$ (num, ch) = STRING$ (num, CHR$ (ch))
```

Эта функция является единственной, которая позволяет задавать параметры в удобном формате.

Пример:

```
100 PRINT TITLE$; STRING$ (50-LEN (TITLE$), " . "); NUM
```

См. также: CHR$ ( ), ASC ( ), VAL ( ), STR$ ( ), STRING$ ( ), SPC ( )

Цель: Обмен значениями двух переменных без использования третьей

Формат: SWAP { *арифметическая переменная*, *арифметическая переменная* } *строковая переменная*, *строковая переменная*

арифметическая переменная :: = любая арифметическая переменная или массив элементов

строковая переменная :: = любая строковая переменная или массив элементов

Действие: Этот оператор позволяет осуществлять обмен содержимым любых двух переменных. Обычно для этого делается следующее:

```
TEMP=A:A=B:B=TEMP
```

— если вы делаете иначе, то это приводит к потере значения одной из переменных. Этот оператор позволяет выполнить обмен за одну операцию — т.е. со значительным увеличением скорости и экономией места. Кроме того, этот оператор можно использовать для работы со строчными переменными и можно произвести обмен указателей (3-х байтных) вместо того, чтобы производить обмен значениями, что делает этот оператор просто отличным по скорости и при этом не занимается области строк и отсутствует "мусор", обычно возникающий при обмене.

Исключительное средство для сортировки строчных массивов. Две переменные, заданные в качестве аргументов должны быть одного и того же типа. Причем, в отличие от других операторов, это требование более жесткое, так, например, нельзя обменивать переменные одинарной и двойной точности. Любое нарушение этого приводит к ошибке "несоответствие типов" (TYPE MISMATCH).

Использование: Этот оператор полезен в любой момент, когда требуется сделать обмен значениями. Однако, наиболее эффективно применение SWAP при сортировке строчного массива. Сравните:

```
100 FOR X=1 TO MAX=1 : FOR Y=X+1 TO MAX
110 IF AR$(X) <=AR$(Y) THEN 140
120 TMP$=AR$(X): AR$(X)=AR$(Y)
130 AR$(Y)=TMP$
140 NEXT: NEXT
```

и программу приведенную в примере. Для массива из 100 элементов по 100 байт каждый, программа занимает только половину памяти необходимую для области строк и требует только 9% времени.

Пример:

```
10 CLEAR 15000: DIM AR$ (100)
20 FOR X=1 TO 100
30 AR$(X)=STRING$(100, 223*RND(1)+33)
40 NEXT
100 FOR X=1 TO 99: FOR Y=X+1 TO 100
110 IF AR$(X) <= AR$(Y) THEN 130
120 SWAP AR$(X), AR$(Y)
130 NEXT: NEXT
```

См. также: Главу 2, раздел Типы переменных.

Цель: Помочь выравнить колонки по оператору PRINT (ПЕЧАТЬ).

Формат: PRINT . . .; TAB (*колонка*) ; . . .

колонка :: = арифметическое выражение в диапазоне от 0 до 255

Действие: Эта специальная функция также, как и функция SPC ( ), допустима только в операторе или команде PRINT. Функция TAB ( ) вызывает перемещение курсора вправо до колонки с указанным номером. По мере перемещения курсора на экран выводятся пробелы. Если значение колонки меньше номера колонки текущего положения курсора, то перемещения не происходит. В этом случае функция TAB ( ) игнорируется. Запомните, что для функции TAB ( ) номера колонок начинаются с нуля: первая колонка имеет номер 0, вторая TAB (1) и т.д. Если выводимая строка занимает несколько строк экрана, то курсор будет перемещаться на последующие строки. Если, например, вы находились в начале строки и ранее был задан оператор WIDTH (25), то TAB (25) в операторе PRINT поместит курсор в начало следующей строки, заполнив предыдущую строку пробелами. Эта функция сообщает оператору PRINT, что от текущей позиции курсора и до колонки с указанным номером надо вывести пробелы.

Использование: Т.к. эта функция управляет выводом, то она не требует дополнительной памяти для хранения выводимых пробелов. Запомните, что в режиме SCREEN 2 или SCREEN 3, если вы выводите данные оператором PRINT на устройство GRP:, то функция TAB вызовет только перемещение курсора, ничего не стирая на экране. В текстовом же режиме (SCREEN 0 или SCREEN 1) до указанной позиции явно выведутся пробелы.

Еще одно возможное использование функции TAB ( ) — это удобное размещение колонок, если вывод по зонам, задаваемым запятой (через 14 позиций), слишком широк. Для специальных целей вы можете также использовать функцию POS (1), которая сообщит вам, где в данный момент находится курсор.

Пример:

```
100 CW=8: WIDTH 28
110 PRINT A; TAB (CW); B; TAB (2*CW); C; TAB (3*CW): D
```

См. также: SPC ( ), WIDTH ( ), POS ( ), CSRLIN ( )

Цель: Возвращает тангенс угла в радианах

Формат: *x* = TAN (*угол*)

угол :: = любое арифметическое выражение

Действие: Эта функция вычисляет с удвоенной точностью тангенс от аргумента, считая, что аргумент — это угол в радианах. Обратной функцией является ATN ( ). Но ATN ( ) не всегда вернет тот же угол, от которого брался тангенс, так как тангенс — функция периодическая. ATN ( ) всегда возвращает угол в диапазоне от 0 до PI/2.

Использование: Эта функция используется для тригонометрических вычислений. Замечания по использованию тригонометрических функций смотри в приложении.

Пример:

```
1000 PRINT "Расстояние="; D; "а угол="; A
1010 PRINT "Поэтому высота="; D*TAN(A)
```

См. также: SIN ( ), COS ( ), ATN ( ), приложение по производным тригонометрическим функциям.

Цель: Возвращает или устанавливает системное время

Формат:
x = TIME
или
TIME = *арвыр*

арвыр :: = любое арифметическое выражение в диапазоне от — 32768 до 65535

Действие: MSX Бейсик имеет внутренний 16-и разрядный счетчик, значение которого изменяется с частотой 50 раз в секунду. Когда значение счетчика достигает 65535, то он сбрасывается в 0 и все начинается сначала.

Если вы используете TIME в выражении, то возвращается текущее значение счетчика в виде беззнакового целого в диапазоне от 0 до 65535. Если же TIME стоит слева от знака равенства, то устанавливается новое значение счетчика. Если TIME присваивается число в диапазоне от 0 до 65535, то присваивание происходит немедленно; если же справа стоит отрицательное число, то оно преобразуется в положительное без знака: —1 это 65535, —32768 это 32768 — и только потом происходит присваивание.

Так как разрядность счетчика невелика, то он обновляется каждые 21.8 минут. Более длительные интервалы времени можно задавать программным путем или с помощью специальной аппаратуры. Очевидно, что каждые 50 тактов — это одна секунда. Отметим, что некоторые операции ввода/вывода как, например, чтение с магнитной ленты приостанавливают работу счетчика.

Использование: Этот счетчик позволяет измерять время для проверки каких-либо временных характеристик. Он также используется для отсчета времени в операторе ON INTERVAL GOSUB, поэтому

вам не нужно делать это программно. Например, при запуске программы вы могли установить TIME=0. Через каждые 10 тактов будет производиться обращение к вашей подпрограмме, заданной в операторе ON INTERVAL. Но, когда, скажем, значение TIME достигнет 600, то прерывание по интервалу будет выключено и вы перейдете к другим действиям.

Запомните: большинство операций ввода/вывода выключают счетчик, и он начинает отсчет заново после окончания ввода/вывода.

Пример:

```
100 PRINT "СЛЕДУЮЩИЙ ВОПРОС:"; QU$
110 TIME=0
120 PRINT "Ответ (1—4)"
130 AN$=INKEY$
140 AN=VAL (AN$): IF AN>=1 AND AN<=4 THEN 200
150 IF TIME>50*10 THEN PRINT "Слишком долго. Вы проиграли!":
    GOTO 10
160 GOTO 130
200 . . . .
```

См. также: ON событие GOSUB, INTERVAL ON/OFF/STOP

TRON / TROFF

оператор

Цель: Включать/выключать трассировку по номерам строк

Формат:
TRON
или
TROFF

Действие: Когда установлен TRON, то номера всех поочередно выполняемых строк выводятся на экран в формате:
[*nnn*]
Курсор перемещается на позицию после этого вывода; если вывод накладывается на какое либо изображение, присутствующее на экране, то оно стирается.
Единственное, что может остановить трассировку — это выполнение оператора TROFF или выключение питания. Эти два оператора можно использовать и в режиме команд. Команда RUN не отменяет действие TRON.

Использование: Если вы создали сложную программу, и она выполняется неправильно, а по выводу на экран нельзя установить, где ошибка, то TRON поможет вам произвести трассировку программы во время ее выполнения. Номер каждой выполненной строки будет выведен на экран, и вы увидите по каким веткам выполнялась ваша программа. Если вам непонятна работа какого-то участка программы, то в начале участка надо поставить TRON, а в конце — TROFF.

Пример:

```
...
249 TRON
250 ' проверяемый участок
...
301 TROFF ' конец проверяемого участка
    [250] [260] [270] [280] [290] [300]
```


фукция

USR

Назначение: Обеспечение доступа к подпрограммам на машинном языке (ассемблере).

Формат: *x* = USR [*цифра*] (*выражение*)

цифра : : = одиночная цифра, 0 - 9. При отсутствии полагается 0
выражение : : = любое выражение любого типа. Должно присутствовать.

Действие: Необходимо задать DEF USR для задания адресов тем 10 функциям к которым допустимо выполнить обращение, в противном случае будет выведено сообщение об ошибке "неверный вызов функции" (Illegal function call).
Если вы определили адрес для функции, к которой хотите обратиться, то указание USR ( ) заставит MSX Бейсик выписать "выражение" - аргумент и обращаться к подпрограмме, написанной на ассемблере Z-80 с помощью инструкции CALL по заданному вами адресу. Подпрограмма может находиться в любом доступном месте памяти, в ключая подпрограммы BIOS MSX Бейсик, находящиеся в постоянной памяти, если только вы не нуждаетесь в установке регистров Z-80 до обращения. Вы можете передать одно значение любого типа вашей подпрограмме, и она, в свою очередь может возвратить одно значение любого типа. Конечно, если тип возвращаемого значения не соответствует ожидаемому в выражении, содержащемуся в USR ( ), то будет выведена ошибка "несоответствие типов" (type mismatch). Вы можете задать до 10 различных подпрограмм.

При обращении программа на ассемблере Z-80 может проверить регистр A для определения типа переданного аргумента. Если содержимое A равно 3, то это строка и пара регистров DE указывает на начало трехбайтного идентификатора строки — первый байт содержит длину строки,

а следующие 2 байта содержат указатель на начало данных строки.

Любое другое значение в A указывает на числовой тип. Подробности смотрите в DEFINT/-SNG/DBL/STR— обычно этот тип 2, 4 или 8, а пара регистров HL содержит указатель на начало числовых данных в формате, требуемом конкретным типом.

Вы можете получить значения любого типа установкой вышеуказанных регистров в соответствующее значение (или оставив их неизменными, чтобы получить значение, выданное вызванной функцией).

После выхода из вызванной подпрограммы, который осуществляется по команде RET микропроцессора Z-80, MSX Бейсик извлекает значение и использует его как результат, возвращаемый функцией USR ( ).

Использование: Важное замечание: не пользуйтесь этой функцией и оператором POKE до тех пор, пока вы полностью не познакомитесь с MSX компьютером. Однако, некоторые возможности данной функции трудно переоценить. Во-первых, можно получить доступ к некоторым возможностям MSX не через Бейсик — например:

```
DEFUSR3=&H003E:x=USR3 ( x )
```

произведет сброс установки всех функциональных ключей к состоянию, имевшему место при включении питания. Но наиболее полезным свойством является возможность, позволяющая вам сделать ассемблерные подпрограммы, выполняющие действия недоступные (или слишком медленные) в Бейсике.

Пример:

```
100 DEF USR = 0: DEFUSR 1 = &HF277 ' FCB 1 буфер 256 байт
110 IF ND = 1 THEN X = USR (X) '
перегрузка по окончании программы
120 PRINT "—>" + USR1 (X$) '
возвращает строчный тип
```

Смотри также: DEF USR, DEF INT/SNG/DBL/STR

Назначение: Выдача числового эквивалента строки символов, содержащей цифры

Формат: *x* = VAL *(strexpr)*

strexpr : : = любое строковое выражение

Действие: Эта функция предполагает, что строковое выражение, его аргумент, содержат допустимое число в формате ASC 11. Это число может включать знак, десятичную точку и запись экспоненты. Эта строка, результат вычисления *strexpr* просматривается слева направо до тех пор, пока не встретится символ, который не может входить в число. Если просмотренные символы представляют собой число, VAL ( ) выдает это число, иначе VAL ( ) выдает нуль. Например, VAL ("a 12:") выдает нуль, а VAL ("23a12:") выдает число 23.

Эта функция противоположна функции STR$ ( ). Детали см. в STR$.

Использование: В большинстве случаев используется в сочетании с STR$ ( ) — для более полной информации см. соответствующий раздел. Однако, VAL ( ) очень хорошо использовать при чтении наряду со строковыми функциями — LINE INPUT a$, INPUT$ (3), INKEY$ и подобные. VAL ( ) позвозляет преобразовать первое встреченное число для математической обработки и т.п. Чтобы найти следующие допустимые числа. которые могут быть в строке, вы можете также использовать INSTR ( ) ; затем используйте MID$ ( ) для выделения этой части, после чего вы можете снова использовать VAL ( ) для преобразования другого числа.

Пример:

```
100 LINE INPUT "Chose 1, 2, or 3:" ; A$
110 ON VAL ( A$ ) GOSUB 300, 400, 500 : BEEP: GOTO 100
```

См. также: STR$ ( ) , INSTR ( ) , LINE INPUT

Назначение: Выдача указателя местонахождения переменных и файлов в памяти

Формат: *x* = VARPTR ([*variable*])
 # file number

variable : : = любая существующая переменная
#file number: : = любой FCB от 0 до MAXFILES

Действие: Эта функция вычисляет и выдает указатели туда, где данные хранятся в памяти. Если вы хотите узнать про номер файла (необязательно открытого). выдаваемый указатель является адресом первого байта блока управления файлом — буфер данных начинается на восемь байтов позже. Если это строковая переменная, указатель является адресом первого байта трехбайтного описателя, который содержит байт длины и двухбайтный указатель данных. Если переменная имеет тип числа, VARPTR ( ) выдает адрес на три байта больше начала переменной. Вычитая три из этого указателя, вы получите действительно начало, которое содержит идентификатор типа переменной.

Адрес выдается в таком формате, что вы можете использовать его в операциях PEEK ( ) и POKE. Если переменная или номер файла не существует, выдается сообщение Illegal function call (неправильный вызов функции).

Использование: Эта функция позволяет быстро и точно определить местоположение любых данных, хранящихся в памяти, и при правильном использовании ON ERROR GOTO, — существует ли эта переменная (все еще). Для модификации данных на свой страх и риск вы можете использовать POKE : например, длина и указатель строковой переменной может изменяться для указания области имени файла в обоасти переменных MSX Бейсик RAM, чтобы любая подпрограмма могла выдать

имя файла, к которому было только что произведено обращение. Или результаты VARPTR ( ) могут передаваться функции USR для извещения этой функции о том, где находятся ее параметры без поиска элементов, например, в целочисленном массиве.

Следует отметить, что любое использование этой функции с определенным элементом массива выдаст вам адрес этого элемента, а не самого заголовка массива — чтобы найти эту информацию, запросите VARPTR (ary (0)) и вернитесь назад (на 3 байта). Неоткрытый FCB также является хорошим местом для засылки в него 256 байтов незащищенных данных.

Пример:

```
100 A ( 1 ) =112:A ( 2 ) =75:A ( 3 ) =VARPTR ( "Testing" )
110 Z$ =USR ( VARPTR ( A ( 1 ) )
```

См. также: USR ( ) , DEF USR ( ) , MAXFILES=

Назначение: Доступ к физическим регистрам VDP (БИС видеодисплея).

Формат:

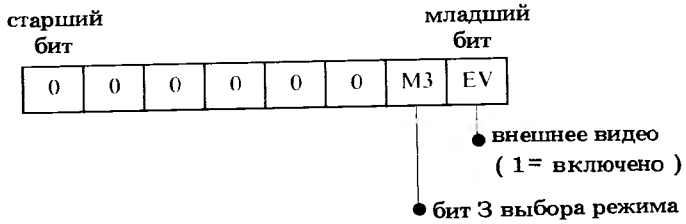
x = VDP *(reg num)*
или
VDP *(reg num) = numexpr*

reg num : : = числовое выражение от 0 до 7
numexpr : : = числовое выражение от 0 до 255

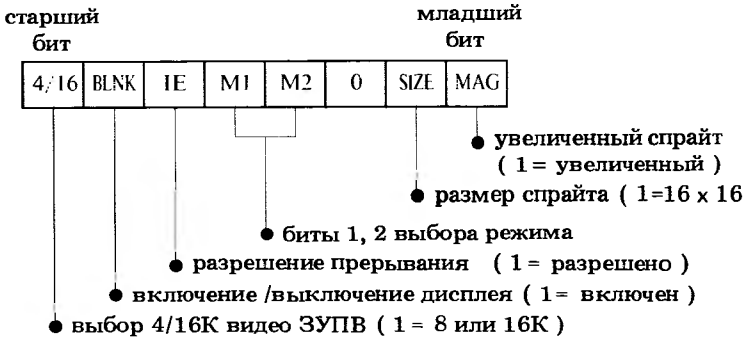
Действие: С помощью этой функции вы можете производить операции чтения и записи в восемь регистров БИС процессора видеодисплея (VDP), совместимого с TI9918A/28A. Они могут исследоваться для физического выполнения того, что BASE ( ) делает неявно: установите режим операции и начало таблиц и считайте регистр состояния, который в числе прочего укажет, произошло ли спрайтное прерывание или пятый спрайт был забланкирован той же самой линией, что и четыре других спрайта.

Информация регистра VDP

Регистр 0



Регистр 1

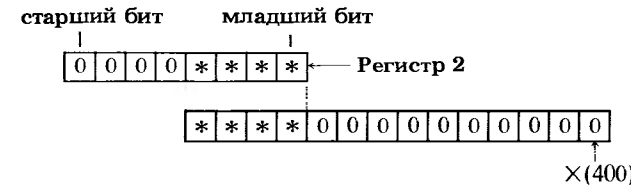


Биты 1, 2, 3 выбора режима

M1	M2	M3	
0	0	0	Графика I (экран 1)
0	0	1	Графика II (экран 2)
0	1	0	Многоцветность (экран 3)
1	0	0	Текст (экран 0)

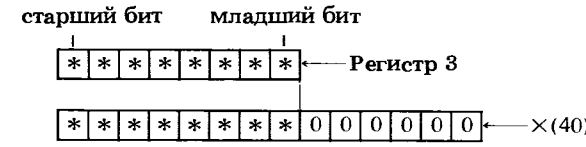
Регистр 2

Таблица названия формы



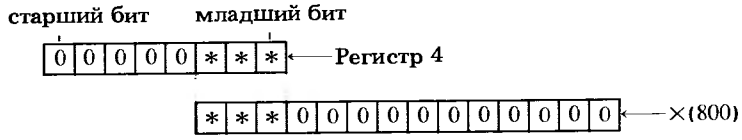
Регистр 3

Таблица цвета



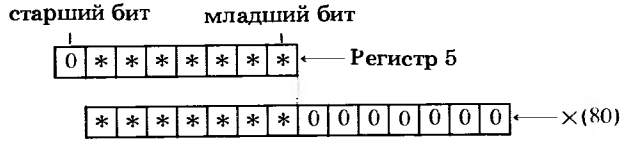
Регистр 4

Таблица генератора формы



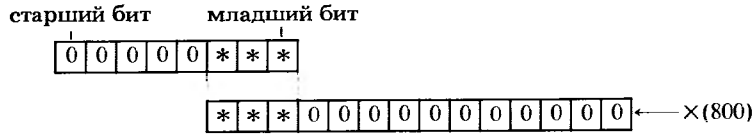
Регистр 5

Таблица характеристики спрайта



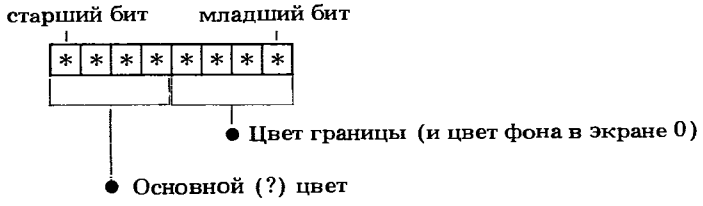
Регистр 6

Таблица генератора спрайта

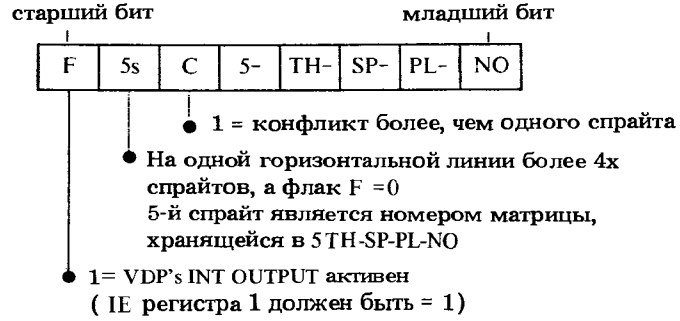


Регистр 7

Цвет



Регистр 8 (только для чтения)



Регистр сбрасывается при чтении или при получении внешнего сигнала сброса.

VPEEK

фукция

Назначение: Выдача байта данных из любого места видео ЗУПВ.

Формат: *x* = VPEEK *(address)*

address : : = числовое выражение от 0 до 16383

Действие: Эта функция выдает байт из любого места 16 к видео ЗУПВ. Эту область, которая полностью отделена от основной памяти, можно прочитать только с помощью инструкций ввода/вывода. Для получения представления о том, каким образом данные размещаются в видео ЗУПВ, см. BASE.

Если выдан запрос адреса, находящегося вне пределов данной области, то происходит ошибка "Неправильный вызов функции". VPEEK ( ) значительно медленнее, чем PEEK из-за способа, которым происходит обращение к видео ЗУПВ.

Использование: Совместно с VPOKE вы можете изменить содержание экрана и спрайта в видео ЗУПВ. Кроме того, вы можете использовать ее для нахождения дополнительного пространства, хранения данных — неиспользуемое видео ЗУПВ является хорошим местом для временного хранения информации, которая в данный момент не нужна (но оно медленнее) .

Пример:

```
100 Z =BASE (7) : S= VPEEK ( Z + 7 )
```

См. также: BASE, VPOKE

фукция

VPOKE

Назначение: Размещение байта данных в любом месте видео ЗУПВ.

Формат: VPOKE *address , data*

address : : = числовое выражение от 0 до 16383
data : : = числовое выражение от 0 до 255

Действие: Этот оператор позволяет разместить один байт данных в любом месте видео ЗУПВ. Если он находится в таблице экранной информации, вы, вероятно, сразу увидите результат; если он в неиспользуемой видео ЗУПВ, то не произвойдет никакого видимого эффекта. В зависимости от того, каким образом обращаются к видео ЗУПВ, за исключением команд ввода/вывода Z-80 это единственный способ разместить данные в видео ЗУПВ и он является более медленным, чем POKE. Видео ЗУПВ полностью отделено от основной памяти.

Если все параметры выходят за данную область, выдается "Неправильный вызов функции".

Использование: Совместно с VPEEK ( ) это дает вам возможность перемещения данных в видео ЗУПВ и обратно. Детали см. в VPEEK.

Пример:

```
100 VPOKE X, VPEEK ( X ) XOR 255
```

См. также: VPEEK ( ), BASE

Назначение: Заставить Бейсик ждать до тех пор, пока порт ввода/вывода не достигнет некоторого значения.

Формат: WAIT *port, and* [, *xor*]

port : : = числовое выражение от —32768 до 65535
xor : : = числовое выражение от 0 до 255
and : : = числовое выражение от 0 до 255

Действие: Оператор WAIT остался от более ранних версий MBASIC. Этот оператор вводит байт данных из указанного порта ввода, применяет операцию XOR с *xor,* а к результату — AND с *and.* Этот процесс повторяется до тех пор, пока полученное значение не станет ненулевым. Если система находится в задержке WAIT, то не происходит никаких системных прерываний (ловушек) ; исключение составляет Control—STOP. Если параметр выходит за допустимые пределы, возникает "Неправильный вызов функции": номер *port* интерпретируется как целое число без знака.

Использование: По существу этот оператор является бесконечным циклом, который ждет, пока от порта ввода Z-80 не придет определенная комбинация сигнал. Выражение *and* указывает каких битов следует ожидать: выражение *xor* может указывать на биты, которые должны стать нулевыми.

Примечание: Номер этих портов может в будущем измениться; поэтому не используйте этот оператор в программах, предназначенных для продажи на рынок.

Пример:

```
100 OUT &HAA, Z: WAIT &HA9, &B00110000, &HFF
100 ' ждет, пока не будет нажат определенный набор ключей
```

См. также: INP ( ) , OUT

WIDTH

оператор

Назначение: Настройка длины текстового экрана

Формат: WIDTH *size*

size : : = числовое выражение от 1 до 32, если SCREEN = 1 или от 1 до 40, если SCREEN = 0

Действие: Этот оператор используется для установки размера текстового экрана, настройки на вывод определенного размера, или учета способа сканирования луча монитора. Он действует только в режимах SCREEN 0 и 1; При первом использовании запоминается и заново используется новая WIDTH, когда этот режим SCREEN устанавливается снова.

Текущая длина по умолчанию запоминается в LINL40, LINL32 и LINLEN: (F3AE), (F3AF) и (F3B0) устанавливается для SCREEN 0,1 и текущего экрана. Когда WIDTH изменяется, экран очищается. Однако, вы можете произвести с подходящими значениями операцию POKE и избежать очистки. Добавляет столбцы слева и справа через раз, поэтому текущее "окно" остается в центре экрана.

Использование: Используется главным образом для настройки экрана на максимально возможное число столбцов, не обрезаемое на той или иной строке вашего монитора. Кроме того, вы можете умышленно расширить размер экрана, вывести некоторый тип надписей, а затем произвести операцию POKE для соответствующих адресов, чтобы сузить экран для защиты ваших надписей.

Пример:

```
100 SCREEN 0: WIDTH 30: SCREEN 1: WIDTH 28
```

См. также: SCREEN

ГЛАВА 4

Руководство по специфике программирования

Эта глава предназначена не для обучения программированию или языку Бейсик. Однако, пользователь, знакомый с основами вычислительной техники, если он хочет понять, как использовать свои знания применительно к MSX Бейсик, найдет в этой главе несколько примеров использования этого языка и описание того, как, различные особенности MSX проявляются в Бейсик-программах.

4.1 Создание программы

Перый вопрос, который задает большинство: -Как сконструировать программу, выполняющую некоторую функцию? В этом разделе даются рекомендации по использванию некоторых методов и типовых приемов.

4.1.1. Определение целей

Большинство начинающих программистов делают ошибку, когда садятся и пытаются записать новую программу "прямо из головы", без предварительного обдумывания и плана. Даже лучшие системные программисты перед написанием программы изображают ее схематически и вот почему: в противном случае любое кажущееся простым изменение может привести к путанице и побочным эффектам, которые потребуют дополнительного анализа. Основное правило для начинающих — сначала подумай — потом напиши.

Любая программа должна состоять из следующих частей:

[ввод] → [обработка программы] → [вывод]

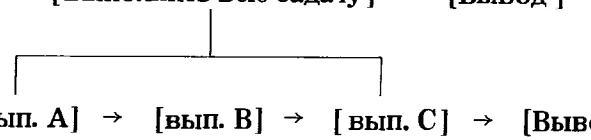
т.е. первое, на что следует обратить внимание, это ясно определить, что небоходимо ввести в программу и что необходимо получить на выходе и в каком формате. Для программы обработка данных — это подразумевает точное описание форматов ввода и вывода. Для игровых задач это означает выявление функций, которые выполняет игровой контроллер и устные и графические сообщения, которые следует вывести на экран MSX. В любом случае: чем лучше описание, тем легче будет

использовать входные и выходные данные. Необходимо учитывать, что физически осуществимо при данном аппаратном обеспечении. К примеру, нельзя выводить в одну строку пятый спрайт, если в ней уже содержатся четыре.

Если вам известны ввод и вывод, необходимо описать процесс работы программы. Короче говоря: что нужно сделать с входными данными, чтобы получить выходные. Для случая программы обработки данных это очевидно, а для игровой программы необходимо постулировать правила игры и возможный ответ на любую из возможных комбинамий входных данных. Чем лучше будет это описание, тем проще будет перевести его на язык программы.

4.1.2. Иерархия и разработка "сверху-вниз".

Предыдущие рекомендации коротко сводятся к следующему: сделайте все полностью. Теперь можно разбить эту возможно непростую задачу на части. Обычно начинают с "выполнить задачу", а затем, разбивая ее на "выполнить А", "выполнить В", "выполнить С" и последовательно производя эти действия, решают всю задачу. В резудьтате мы более детально описываем всю задачу:



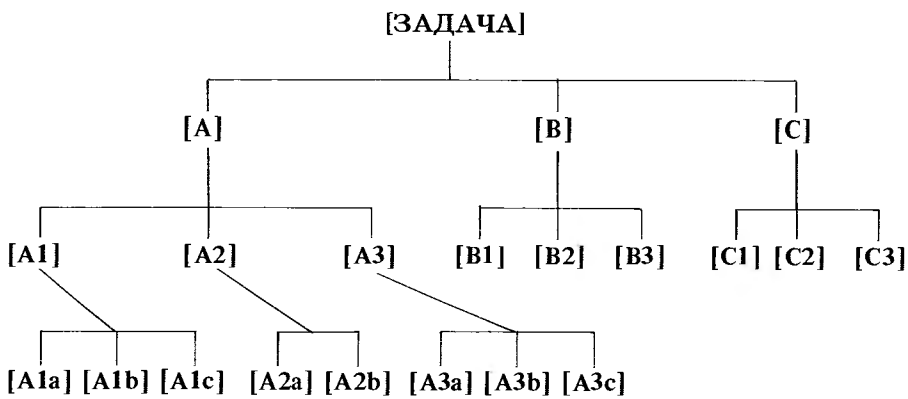
В результате мы имеем общее описание всей задачи, разбитое на несколько частей. Действие этих частей, взятых как целое, соответствует полному описанию, но каждая из трех подзадач намного легче поддается программированию.(Конечно, описание каждого блока соответствует его функциям, и сокращение "Выполнить А" приводится лишь для иллюстрации).

Если проследить за потоком данных со входа к обработке и к выходу, можно заметить, что на детализированном уровне он такой же, как и на верхнем: каждый блок в действительности описывает, что поступает на вход, какая происходит обработка данных и что получается на выходе. Совершенно логично

определять блок следующим образом: выполнять В пока нажата 1-я кнопка рычажного указателя. В действительности этот тип конструкции так же является логическим как и FOR LV = 1 то 10 (т.е. выполнить этот блок 10 раз).

Заметим, однако, две вещи: никогда не следует "возвращаться назад", и следует применять правила "повторения" ко всему блоку как к целому.

Что, если какой-нибудь блок, например, "Выполнить А", все еще является блоком "высокого уровня", т.е. может быть далее разбит на части? В этом случае следует продолжить процесс детализации. Каждый из блоков следует разбить на ряд более детализированных, точно так же, как верхний блок был разбит на 3 подблока. Этот прцесс может быть продолжен до тех пор, пока каждый из блоков можно будет легко перевести в несколько операторов на языке Бейсик.
Например:



Заметим, что по необходимости можно расширить, сузить этот ряд или оставить его без изменений. К примеру, последний нерасширенный блок раскрывается ниже в виде последовательности блоков. Таким образом, каждая законченная последовательность блоков представляет собой целую программу, обозначенную [задача].

Кроме того, поток данных от входа к выходу, слева направо представляет собой вход в каждый из блоков, обработки в нем и выхода, причем каждый блок получает инфорацию из блока [ввод] и с выходов предыдущих (расположенных слева от него) блоков.

Если бы вам удалось все это выполнить, то у вас будет такое точное описание, что написание соответствующей программы не составит труда. Вы уже учли множество проблем программной логики, которые так усложняют отладку.

Следует принять во внимание два следующих фактора.

Во-первых, что делать, если имеется одна или более независимых подпрограмм — таких как подпрограмма читки данных, которая разблокирует и возвращает следующий правильный байт данных после декодирования и преобразования и используется для получения байта во многих различных процедурах на протяжении всей программы? В этом случае можно разбить подпрограмму на нижнем уровне схемы (можно на несколько уровней). Теперь, если это необходимо, проведите пунктирную линию от нужного места на диаграмме вниз к подпрограмме.

Во-вторых, как выполняются ручные прерывания? В MSX Бейсик существует несколько условий, которые могут привести к "асинхронным" прерываниям программы и вызову подпрограммы обработки прерываний. Эти подпрограммы прерываний, характеризуемые ON ситуациями GOSUB nnnn, должны быть расположены в стороне (с несколькими возможными уровнями детализации). Следует позаботиться о том, чтобы они не оказывали влияния на основную программу, в которой могла возникнуть ситуация прерывания, а также учитывать, какое именно прерывание допустимо (до какой степени возможно прерывание в то время, как подпрограмма обрабатывает другое прерывание)!

Запомните, что этот контроль является частью вашей основной программы, и следует описывать ON и OFF ситуации внутри основной программы вместе с другими описаниями для того, чтобы контролировать эффективность обработки этих прерываний.

Описанный выше анализ основной программы носит название HIPO-метода (Иерархия Вход/Обработка/Выход) и является одним из лучших методов программирования "сверху-вниз". Почему "сверху-вниз"? Просто потому, что вы начинаете сверху — с общего описания программы и развиваете его шаг за шагом, проясняя и детализируя каждый шаг.

4.1.3. Написание программы.

Теперь можно начинать программировать. На самом деле, если ваше описание задачи соответствует представляемому выше, вы, повидимому, можете сэкономить время и сразу сесть за клавиатуру вместо того, что записывать программу вручную. (Многие поддаются искушению и приступают к этому еще раньше).

Во-первых, вы должны слелать следующее: отведите под каждый блок в вашей схеме последовательность номеров. Предположим, 200 номеров на каждый блок. По крайней мере это допустимо для трех различных блоков. Старайтесь сделать это логично и используйте идентичную, скажем, двухразрядную приставку для каждого из уровней (последовательностей блоков), либо какую-нибудь другую разумную. Например, номера: 00200 - 00399 = A, 00400 — 00599 = B, и 00600 — 00799 = C: затем: 10000 — 10199 = A1, 10400 - 10599 = A3, 11000 — 11199 = B1, 11400 — 11499 = B3, 12000 - 12199 = C1, 20000 — 20199 = A1a:, т.д.

На самом деле можно просто присвоить каждому из блоков любые 200 последовательных номеров, однако, окончательный вариант должен помочь вам сэкономить время на поиск соответствующей последовательности номеров при отладке программы.

После того, как нумерация закончена, можно писать саму программу. Каждый блок, особенно в верхних слоях диаграммы, обычно оформляется следующим образом:

```
00100   'Main Program [... comments...]
00150   GOSUB 00200' Do A
00160   GOSUB 00400' Do B
00170   GOSUB 00600' Do C
00180   END' end of the program

00200   'Routine to Do A [... comments ...]
00210   GOSUB 10000' Do A1
00220   IF xx = 1 THEN GOSUB 10200: GOTO 00220' While xx=1
        Do A2
00230   GOSUB 10400' Do A3
00240   RETURN' from Do A
```


..... и т.д.

Заметим, что каждая подпрограмма, как правило, состоит из последовательности вызовов других подпрограмм. Здесь же кроется ответ на вопрос: Каким большим может быть кождый из блоков? Ответ: Таким, чтобы он полностью помещался на экране дисплея. При этом его можно будет просмотреть весь сразу. Теперь можно ответить и на другой вопрос: что следует расположить в каждом блоке? Ответ: Если этот блок не находится на нижнем уровне, то он обычно представляет собой последовательность из GOSUB -ов, снабженную, возможно, логическими условиями выполнения GOSUB-ов. Вы будете удивлены, насколько простыми окажутся блоки нижнего уровня.

Вы пока не в состоянии детально описать блоки нижнего уровня? Тогда при наличии свободного экрана или печатающего устройства, вы можете заменить эти блоки чем-то вроде:

```
[L] PRINT "You just did XXXXX" : RETURN
```

и после этого начать отладку вашей программы. Теперь вы можете убедиться, что этот блок действительно вызывается в нужный момент.

Каждая из созданных вами подпрограмм — одна на каждый блок должна иметь следующие характеристики: только одну точку входа и только одну точку выхода (RETURN в конце подпрограммы). Кроме того, любая переменная, используемая только в этой подпрограмме, доджна быть соответствующим образом и описана, чтобы в дальнейшем вы бы случайно не использовали ее в другом месте. Так же внимательно следует записать все переменные, используемые как в этой подпрограмме, так и во всех подчиненных подпрограммах (более детализированного уровня).

Никогда не используйте в подпрограмме переменные, которые описаны как локальные в каком-либо другом месте! Разделяйте визуально подпрограммы строками пробелов или комментариев. Так вам будет легче.

4.1.4 Документирование текста программы.

Многие целиком пренебрегают комментариями к программам или приписывают бездумные сообщения к готовой программе. Они говорят, что комментарии только занимают лишний объем памяти и увеличивают время счета. Это так, но вы всегда можете удалить их из программы, создав тем самым ее версию только для счета и выиграв в скорости и объеме памяти. Они скажут, что и так знают все, что происходит в программе. И это, возможно, верно, но через несколько недель что-нибудь забудется. Они скажут, что им никогда не придется модифицировать конечный продукт. Это грубая ошибка. Неожиданная необходимость в добавлениях и изменениях каких-то особенностей вашей программы может привести к хаосу и недоразумениям, когда, к примеру, забыто назначение оператора "GOSUB 32" либо неизвестно зачем в этом месте программы обнуляется некоторая переменная.

Ответ прост. Вам следует всегда, с начала до конца, документировать текст вашей новой программы во время ее создания. Для любого значительного проекта, — в этом заключается одна из разумных и необходимых возможностей застраховать себя от непредсказуемых заранее неприятностей.

Существует три уровня документирования, которых следует придерживаться. Первый уровень — это общее описание программы в начальном блоке комментариев. Он включает в себя даты создания и последующих модификаций, номер версии, имя автора и авторские права (если это необходимо). Здесь же следует указать цели разработки, задав вход, обработку и вывод. Кроме того, следует описать используемый алгоритм: шаг за шагом, что делает каждый блок вашей программы (в том случае, если вы не произвели полную разработку вашей программы по предложенной выше схеме). Эта часть может послужить в качестве введения для неосведомленного, что делает ваша программа и как ею пользоваться. Большинство разработчиков экспертных программ считают, что лучшим способом написания программы является предварительное написание к ней учебного руководства.

Второй уровень документирования заключен в каждом из блоков программы и/либо в каждой подпрограмме (если вы применяете приведенные выше методы проектирования). В

каждом блоке программы следует отметить его начало и объяснить, что это "шаг xx", который соответствует таким-то шагам, описанным ранее в алгоритме. Для подпрограмм пометку их начала следует снабжать комментариями, объясняющими, что делает эта процедура, какие переменные вводятся и какие выводятся из процедуры, какие переменные локальные и (если это необходимо) объяснение алгоритма.

Наконец, третий уровень документирования — построчный. Конечно, необходимо пояснять любую необычную комбинацию операторов или любой хитроумный прием программирования. Например, какую функцию выполняет VPOKE BASE (7) * 10,55 и почему вы использовали в этом месте этот оператор. Кроме того, когда бы вы ни использовали GOSUB или GOTO или что-нибудь в этом роде, удостоверьтесь и сформулируйте на словах, что вы собираетесь сделать. Цель заключается в том, чтобы, прочтя комментарии к подпрограмме, которая может просто состоять из четырех GOSUB и RETURN, вы могли бы точно ответить на вопрос — что делать дальше? Приведенный выше листинг программы может служить примером (конечно, содержание комментария Do A1 — только иллюстрация).

Если вы будете придерживаться этих рекомендаций, то время, что вы потратите теперь на набор более длинного текста программы, окупится через полгода тем, что вы будете понимать ее.

4.2 Особенности графики.

Наличие превосходной машинной графики — одна из замечательных особенностей MSX Бейсика и MSX вообще. Быть может, некоторые машины на рынке ЭВМ обладают большими возможностями в других областях аппаратного обеспечения, но в области машинной графики другой такой машины не существует.

4.2.1. Общие возможности.

MSX — компьютер, снабженный дополнительным процессором TI VDP, который удовлетворяет потребностям графики. Однако,

это аппаратное обеспечение является ограничивающим фактором. Этот процессор имеет свою собственную память — 16 К с произвольной выборкой, которая полностью независима от основной памяти компьютера. Различные режимы работы обуславливают различное использование этой памяти и умелое обращение с неиспользованной видеопамятью с произвольной выборкой может дать дополнительные преимущества.

Видеопроцессор (VDP) имеет четыре основных режима вывода изображения на экран, обозначаемых на языке Бейсик как SCREEN 0, 1, 2 и 3. 0 и 1 — текстовые режимы; 2 и 3 — графические режимы. Это означает, что стандартный вывод текста, например, с помощью операторов PRINT, либо все, что печатается при выводе возможно только в режимах 0 и 1. При печати текста в номер файла, предварительно открытого, как устройство "GRP:", можно в ограниченном виде вывести текст, находясь в режимах 2 и 3, но все, что уже было перед этим, будет напечатано вверху: никакой из имевшихся знаков не будет удален. Точно так же вовсе нельзя пользоваться операторами графики, такими, как PSET или CIRCLE, находясь в текстовом режиме.

Для того, чтобы видеопроцессор (VDP) выдал вам состоящий из 40 колонок текст в режиме SCREEN 0, необходимо следовать нескольким правилам: никаких спрайтов, никакой раздельной граничной раскраски, только один цвет на переднем плане и один цвет на заднем плане для всего экрана в любое время. Кроме того, операторы изображаются максимум в шести столбцах слева сразу за первыми восьмью столбцами, в которых располагаются их номера.

В режиме SCREEN 1, ширина текста составляет 32 колонки. Вы можете независимо изменять цвета с помощью оператора COLOR, который устанавливает все цвета переднего, заднего плана и границ.

В каждом из этих режимов оператор COLOR устанавливает начальный цвет всего экрана; оператор WIDTH — ширину текста, который центрируется; функции POS ( ) и CSRLIN возвращают положение курсора; оператор LOCATE устанавливает положение и состояние курсора; оператор PRINT обычным образом выводит данные на дисплей.

В графическом режиме SCREEN 2 пользователю доступны 256 x 192 пикселей строк, как в режиме SCREEN 1, но каждый пиксель является индивидуально доступным. В этом режиме каждая строка экрана делится на группы из 8 элементов. В каждой группе в таблице цветов соответствует один байт, который определяет основной и фоновый цвета (1 из 16 возможных для каждого из них). Кроме того, в другой таблице-в таблице изображения на каждый пиксель отводится один бит, задающий основной или фоновый цвет элемента. Следовательно, любая группа из 8 элементов может иметь только два цвета; попытка изобразить третий цвет изменит цвет уже существующих точек. При тщательном планировании с учетом этих ограничений можно составить детальное и многоцветное изображение. Важная особенность системы MSX-Бейсик заключается в том, что каждая команда графики рассматривается как изменение цвета изображения при сохранении фонового цвета с момента последнего использования оператора COLOR; однако, для заполнения больших областей некоторым цветом обычно делают этот цвет составной частью фона, как в операторе LINE ( ) – ( ) ,, BF.

Применение этого приема требует внимательности, но в целом результаты получаются следующие: если операторы PAINT или LINE ( ) – ( ) ,, BF записывают одинаковый цвет в байт, соответствующий группе из 8 пикселей изображения, то 1) весь байт таблицы изображения устанавливается в ноль (признак фона); 2) задается требуемый заполняющий цвет, который становится цветом фона в заполняемой области и 3) цвет изображения не изменяется. Отметим, что с помощью других операторов, например, оператора LINE ( ) – ( ) ,, BF, такое заполнение группы из 8 элементов изоброжения невозможно.

Отметим, что если при выполнении оператора, устанавливающего точку в группе из 8 элементов изображения, требуется изменить только один нулевой (фоновый) бит в байте таблицы изображения и цвет точки не совпадает с цветом изображения, или же если все биты в байте таблицы изображения уже установлены в цвет изображения — (то есть, все 1), тогда, как и в ранее описанном примере, указанная группа битов таблицы изображения устанавливается в 0, цвет фона изменяется, а цвет изображения не изменяется.

Во всех остальных случаях указанный бит устанавливается в 1

(цвет изображения), сам цвет изображения соответствующим образом изменяется, а фоновый цвет остается прежним, даже если эта приводит к тому, что остающиеся 7 битов цвета изображения в этом байте таблицы изображения будут задавать неверный цвет: ранее установленные в 1 биты этого байта, цвет которых переопределяется в соответствии с цветом, приписанным вновь установленному биту, сохраняет свое значение, а значит, изменяет свой цвет! Если вы дочитали этот абзац, советуем передохнуть.

Рассмотрим теперь организацию таблицы цвета и таблицы изображения в режиме SCREEN 2. Напомним, что каждый байт соответствует 8 смежным пикселям изображения: в одной группе первых 8 байтов соответствуют первым восьми элементам изображений первых 8 линий, следующие 8 байтов соответствуют вторым 8 элементам первых восьми линий и так далее вправо до 32 групп из 8 байтов для описания цветов первых восьми строк изображения. 33-я группа из 8 байтов начинает описание цветов для 9 — 15 строк изображения аналогично описанию первых восьми строк и т.д.

0	8	16	24	...
1	9	17	25	
2	10	18	26	
3	11	19	27	
4	12	20	28	
5	13	21	29	
6	14	22	30	
7	15	23	31	
256	264	272	280	
257	265	273	281	
258	266	274	282	
⋮				

Соответствие нумерации байтов таблицы изображения элементам самого изображения, состоящим каждый из 8 пикселей одной строки, показано на рисунке.

Далее, с каждым из этих элементов изображения связан как байт таблицы изображения (по 1 биту на один пиксель этого элемента изображения), так и байт таблицы цвета, указывающий фоновый и основной цвета. Форматы этих таблиц совпадают. Следует отметить сходство этого формата с определением набора символов (по 8 x 8 элементов изображения на символ), если формат кадра — 24 строки по 32 символа в каждой. В этом режиме, как и в текстовом, используется также таблица имен символов, в которой указывается, какой по счету символ из полного набора (256 символов) нужно вывести на указанную позицию экрана.

На экране существует 3 x 256 полей для изображения символов, которые распределены по трем окнам (верхнее, среднее и нижнее), каждое состоит из 256 полей. Соответствующие три набора байтов размещаются один за другим без разделителей, если используются таблицы цвета и изображения. Существует таблица, которая инициализирована так, что содержит байты от 0 до 255, повторенные три раза, т.е. первая треть соответствует первым восьми строкам и т.д. Это помогает путем перемещения одного байта таблицы изменить 64 элемента изображения на языке Бейсик.

В режиме SCREEN 3 существует только 64 элемента изображения на каждой из 48 строк экрана и каждый элемент имеет номер цвета от 0 до 15. Следовательно, побочные эффекты отсутствуют и имеется возможность окрасить с помощью опратора PAINT некоторый участок, ограниченный многоцветным контуром, что нельзя сделать в режиме SCREEN 2, который требует для закраски одноцветного контура. Этот режим полезен тем, что (а) требует только 2К VRAM на экран, (б) разрешает использование готовых фрагментов изображения, что нельзя в режиме SCREEN 0, который также требует небольшого объема VRAM,и (в) можно открыть "GRP" и выводить на экран буквы большого размера. Недостаток режима SCREEN 3 заключается в том, что координаты идентичны режиму SCREEN 2 и точность уменьшается в 4 раза, т.к. точки от (0,0) до (3,3) описывают один элемент изображения. Запись в одну из этих точек изменяет весь элемент изображения. Это позволяет легко перемещать программы между графическими экранами.

Большой проблемой при использовании графических экранов является вывод текста. С помощью устройства "GRP:" можно

вывести текст, но как записать вторую строку в той же самой области? Существует 3 способа:

(а) очистить область с помощью оператора VPOKES перед перезаписью, (б) очистить область с помощью оператора LINE ( ) – ( ), BF перед перезаписью (в) очистить область с помощью оператора меняя основной и фоновый цвета, точно "перепечатать" первоначальный текст, затем с помощью оператора COLOR опять поменять основной и фоновый цвета и вывести новый текст. Текст позиционируется так, что первый симвод размещается с верхней левой точки позиции графического курсора. Следовательно, оператор PRESET (x, y) может разместить начало текста (предполагается, что точка является фоновой). Тескт будет восстановлен и никакой символ не будет "прижат" к краю экрана; оператор WIDTH игнорируется. Все символы представляются матрицей 8 x 8 (как в режиме SCREEN 1), дающей 32 возможных символа, но только центр 30 имеет большую вероятность не оказаться прижатым к краю. При невнимательности легко "размазать" текст из-за наложения. Так как текст является просто совокупностью точек на экране, то при размещении символов с использованием вышеизложенных правил наложение не возникает. Отметим, что оператор PRINT #1, "␣" – вывести один пробел не вызывает никакого действия и не изменяет экрана. Такой эффект иногда может быть полезен.

4.2.2 Операторы для изображений с высоким разрешением.

Графические изображения с высоким разрешением реализуются в режиме SCREEN 2. Хотя большинство следующих операторов работают и в режиме SCREEN 3, но в этом режиме ими пользуются реже.

Для того, чтобы нарисовать прямую линию, используется оператор LINE. Для изображения прямоугольника (или его контура) также используется оператор LINE. Для окружностей, дуг и секторов используется оператор CIRCLE. Для отдельных точек используется оператор PSET, а оператор COLOR определяет цвет, формируемый по умолчанию. Для сложных рисунков используется компактный графический макроязык (GML). GML дает возможность указывать повороты с дискретностью по 90 градусов и масштабировать изображения, создаваемые с помощью оператора DRAW. Для того, чтобы закрасить область,

охваченную произвольным контуром, используется оператор PAINT. Отметим, что оператор PAINT в SCREEN 2 заполняет контур только тем же цветом, который имеет граница контура. Контур другого цвета можно получить только задавая этот контур повторно. В режиме SCREEN 3 такой проблемы не существует.

Система координат является достаточно гибкой. Во всех графических операторах, за исключением оператора PAINT, можно адресовать пространство (–32768 : 32767, –32768 : 32767) . Координаты увеличиваются вправо и вниз. Верхний левый угол экрана является точкой (0, 0) ; изображение внутри окна (0:255, 0:191) будет видно на экране, остальное отсекается без потери скорости, (за исключением оператора CIRCLE) . Меняя координаты, легко изобразить рисунок много больший, чем кадр на дисплее (это возможно с помощью масштабирования) и разместить изображение так, что любой желаемый участок его будет показан на экране без специальных вычислений.

Любое изображение можно задать в абсолютных и относительных координатах. В первом случае указываются фактические значения X и У (столбец и строка растра на экране) . Во втором случае с помощью оператора STEP указывается смещение от последней специфицированной точки. Таким образом, подпрограмма, которая рисует дом, может вызываться несколько раз подряд; дома появятся в разных местах экрана. Кроме того, определяется масштаб изображения.

Все графические операторы запоминают координаты перемещающегося "графического курсора" по последней "нарисованной" точке (исключение составляет операторы печати текста) . Например, оператор LINE (a, b) – STEP (c, d) оставляет графический курсор в последней точке – в этом случае (a + c, b + d) . Некоторые команды разрешают использовать координаты этого курсора не только как точку отсчета при работе в относительных координатах, но и как неявно задеваемую начальную точку. Оператор LINE-(a, b) нарисует линию от графического курсора до точки (a, b) .

Любые графические операторы будут использовать основной и фоновый цвета, заданные в последнем исполненном операторе COLOR . Эти цвета могут быть изменены при выполнении любого графического оператора.

Оператор CIRCLE имеет самые сложные параметры, особенно параметры начала и конца дуги. Этими параметрами являются углы в радианах, измеренные по часовой стрелке от линии, направленной вправо. Например, PI/4 обозначает 45°; PI/2 — прямой угол; PI —линию, направленную влево (приблизительно, 0,785,1.57 и 3.14 соответственно — большая точность не нужна). Таким образом, указываются начальная и конечная точки дуги. Если каждая из этих величин является отрицательной, то изображается такая же дуга и радиус, соединяющий конечную точку с центром. Другой параметр обозначает степень овальности. Время выполнения оператора CIRCLE приблизительно одинаково и не зависит от величины изображения на экране; меньшее время требуется при меньшем радиусе.

В заключение отметим, что оператор PAINT работает медленно, но оператор LINE ( ) — ( ),, BF закрасит прямоугольник быстро. Медленно закрашиваются большие рисунки (чем меньше, тем быстрее), медленно изображаются круги большого радиуса и не существует BF —версии оператора CIRCLE,но если необходимо закрасить круг и его центр размещается на экране дисилея, то после оператора CIRCLE необходимо выполнить оператор PAINT STEP (0, 0), что закрасит круг.

4.2.3 Фрагменты изображения.

Наличие перемещаемых фрагментов изображения — наиболее полезная особенность графики языка Бейсик MSX. Если необходимо просто изобразить некоторую комбинацию 8 x 8 в фиксированных точках, то определяется символ, который будет наиболее подходящим, а фрагменты изображения могут быть размещены где угодно без разрушения образа с помощью изменения одного байта. Так же может быть изменен цвет фрагмента. Кроме того, имеется возможность для каждого фрагмента иметь несколько вариантов и изменение одного байта будет вызывать изменение фрагмента в момент его изображения.

Существуют 32 фрагмента. Каждый имеет 4 характеристики: цвет, позицию (X, У) и номер варианта. Существует два общих параметра: размер фрагмента (нормальный или двойной) и длина изображения фрагмента (8 байтов для 8 x 8 элементов изображения или 32 байта для 16 x 16 элементов). Эти два

общих параметра должны быть установлены вместе с оператором SCREEN до определения любой другой информации фрагмента; такое действие в языке Бейсик будет вызывать стирание всех данных фрагментов. Отметим, что фрагменты не используются в режиме SCREEN 0.

Существуют специальная зона VRAM размером 2К, зарезервированная для данных фрагментов, и порядок выборки фрагмента, т.к. данные этой зоны сами не являются фрагментами изображений. Действительно, все 32 фрагмента могут использовать один и тот же образ. Часто один образ используется в одном фрагменте, но просто переназначить любой из образов любому фрагменту, таким образом изменяя его внешний вид, но не положение или цвет.

Т.к. фрагменты могут иметь только один цвет каждый, то единичные биты в изображении обозначают цветной фрагмент, а нулевые прозрачный. Если используются фрагменты 8 x 8, то каждый байт соответствует образу (всего 256) и каждому байту соответствует 8 элементов изображения, которые имеют такой же размер, как в режиме SCREEN 2. Если используются фрагменты 16 x 16, то место в VRAM существует только для 64 фрагментов; 4 последовательных фрагмента 8 x 8 комбинируются следующим образом, чтобы изобразить большой фрагмент.

фрагмент 1 : фрагмент 3

фрагмент 2 : фрагмент 4 — порядок появления на дисплее

Данные могут быть помещены в таблице фрагментов с помощью псевдопеременной SPRITE$ ( ). Она также обеспечивает выборку данных. Псевдопеременная содержит информацию о числе битов, имеющихся в записи фрагмента, а также номер ячейки VRAM, с которой начинается его размещение. Нумерация фрагментов в псевдопеременной VRAM начинается с нуля. Если вам не нужно воспроизводить фрагменты, то информацию о них можно разместить за счет уплотнения ваших данных, когда область оперативной памяти заполнена. Вы должны ввести характеристические данные в SPRITE$ ( ) (см. DATA для особых приемов).

Как только вы определили образ с помощью выражения (оператора) SPRITE$ ( ), вы можете задействовать его отображение с помощью оператора PUT SPRITE x. В этом выражении x

является номером отображаемого фрагмента (0—31), но не номером образа. Используя данный номер для первоначального описания позиции, цвета и выявления фрагмента, связанного с ним, впоследствии снова можно применять PUT SPRITE для изменения всех или только некоторых из этих характеристик.

В отображении фрагмента вы можете путем некоторых перекрытий эффективно получить разноцветное изображение или же расширить его масштаб. Между тем, здесь имеется проблема.

Вы лимитированы четырьмя отображаемыми характеристиками на каждой линии экрана. Любые дополнительные данные исключаются (только на этой линии). Сюда относятся даже прозрачные части фрагмента. Вы можете использовать VDP ( ) для определения (выявления) первого фрагмента, который затемняется таким путем. Самый последний номер фрагмента, который отображается, тоже четвертый, но не первоначально намеченный четвертый образ.

Используя фрагменты размером 16 x 16 и вводя 2х кратное увеличение, вы можете реализовать фрагменты размером 32 x 32 пикселей (точек), но имейте в виду, что вместе с тем каждый элемент фрагмента при этом имеет размер 2 x 2 экранных точек. Размещая рядом вплотную друг к другу четыре фрагмента, можем занять половину ширины экрана дисплея и более, чем достаточное число фрагментов повторяет этот эффект на высоте экрана.

Состояние отображения дешифрируется с помощью выражения ON SPRITE GOSUB / SPRITE ON. Всегда два фрагмента перекрываются, исключая перекрытия для их прозрачных частей — прерывание, если оно вводится (принимается). Если вы желаете воспользоваться этой возможностью, не перекрывайте изображений для полчения разноцветных эффектов. Кстати, MSX Бейсик всегда вызывает подобное прерывание с помощью выражения PUT SPRITE, но при этом нет возможности определить, какой фрагмент был сдвинут и вызвал это (прерывание) и нет возможности удалить фрагмент, "столкнувшийся" с ним. Удобным путем для нахождения первого из них является применение выражения PUT SPRITE вида:

```
sp = 12: PUT SPRITE sp. . . . .
```


Тогда ваша программа прерывания фрагмента сможет зафиксировать переменную sp с тем, чтобы определить текущее состояние. Любая дополнительная информация должна быть собрана либо путем точного сохранения вашей собственной таблицы, где содержатся все фрагменты, либо путем получения таблицы с помощью оператора VPEEK ( ) к 128-битной таблице атрибутов спрайта и вычислений, которые должны накладываться друг на друга.

Наиболее интересный трюк, с помощью которого можно преодолеть ограничение "4 спрайта в строке", это "отодвигание" неиспользуемого спрайта на задний план и "выдвижение" спрайта снова на передний план только в том случае, когда его необходимо переместить. Это особенно легко осуществить для спрайтов, расположенных по вертикали, т.к. количество строк экрана кратно 8, и тем более легко, если размер спрайта равен 8 x 8. Для передачи шаблона спрайта, заданного оператором SPRITE$ ( ) „ в таблицу шаблона для оператора SCREEN 2, надо использовать операторы VPEEK и VPOKE.

Если вы хотите плавно перемещать спрайт для получения эффекта мультипликации, то используйте операторы цикла FOR-NEXT. Или же можно, создав соответствующую подпрограмму обработки прерываний, использовать оператор прерывания по таймеру ON INTERVAL GOSUB, и в этой подпрограмме выполнять оператор PUT SPRITE.
Если вы установили TIME в ноль, то это можно использовать в качестве счетчика цикла.

4.3 Возможности работы со звуком.

Не так уж много домашних компютеров, которые обладают возможностями воспроизводства звуков в MSX Бейсик. Имеется два пути синтеза звука: автоматический (с PLAY и BEEP) или ручной (с SOUND).

4.3.1. Автоматическая генерация музыки.

Наиболее простым способом вывода звука является сигнал

BEEP. Он останавливает и прекращает любой вывод текущего звука, выдает короткий сигнал тревоги и устанавливает в начальное состояние все регистры SOUND и PLAY. BEEP дается автоматически при получении сообщения об ошибке, нажатии контрольной клавиши STOP или при выполнении оператора STOP.

Более интересным является музыкальный макроязык. используемый оператором PLAY. Это дает вам возможность одновременного описания трех исполняемых нот и может установить такую очередность, что музыка будет автоматически исполняться до тех пор, пока Бейсик будет выполнять последующие операторы. Функция PLAY ( ) способна также выявить, звучит ли уже фоновая музыка.

Каждый из трех голосов имеет свои независимо вводимые характеристики, которые используются в синтезируемой музыке. Они включают частоту и форму сигналов, силу звучания, длительность пауз, октаву и темп. Заметьте, что частота, форма и сила звука — взаимно исключающие возможности! Итак, вы можете описывать ноты с помощью чисел или октав, букв с возможным наложением их длительностей. Комбинирование всем этим позволяет получить очень приятную музыку.

Проблемы? Во-первых, когда без остановки исполняется длительная последовательность музыкальных звуков для трех голосов, то можно выйти за границы последовательности. Для устранения этого недостатка необходима быстрая остановка исполнения музыки для восстановления синхронизации голосов.

Когда оператор PLAY выполнен, то, пока MML устанавливает очередность нот, возникает небольшая задержка. Два подряд стоящих оператора PLAY не могут быть заданы так, чтобы выдержать исполнение одних и тех же музыкальных нот без небольшой паузы. Если оператор PLAY оказывается очень продолжительным, то при исполнении возникнет пауза, пока не освободится пространство, позволяющее поместить в очередь остаток исполняемой мелодии до выполнения следующих операторов. Поскольку PLAY ( ) сообщает об окончании исполнения мелодии, то очень трудно обеспечить непрерывность мелодии, если одновременно выполняется что-либо еще. Наконец, единственный путь избежать полного легато при исполнении музыки (так PLAY "CC" будет звучать подобно двум нотам) — это ввести общее стакатто с контролем частоты и формы, но при

этом теряется управление силой звука. Альтернативой является сокращение каждой ноты и добавление короткого остатка между нотами, что очень трудно и расточительно с точки зрения используемого пространства. но этот вариант работает.

4.3.2 Прямой доступ к выводу звука

Имеются два пути к прямому доступу для вывода звука. Во-первых, имеется побитовый вывод звука, достигаемый через бит 7 порта C PPI. Вы можете также использовать оператор OUT, но эффект, за исключением использования машинных подпрограмм будет незначительным.

Основной способ воздействия на вывод звуков без использования стандартных нот — это использование оператора SOUND. Он позволяет изменять значения регистров PSG. В этом случае вы можете непосредственно создать много звуковых эффектов, включая использование генератора шума и генерацию сигналов сложной формы. Смотрите описание оператора SOUND, где описано использование регистров. Использование оператора SOUND может пригодиться для многих программ.

4.4 Операторы ввода-вывода

В дополнение к таким операторам ввода-вывода, как PRINT, INPUT в MSX Бейсик имеется целая система файло-ориентированных операторов ввода-вывода. К трем уже имеющимся именам устройств в/в MSX Бейсик позволяет добавлять имена других устройств, которые могут быть подключены в дальнейшем с использованием кассет ПЗУ.

4.4.1 Ввод с клавиатуры

Во-первых, одно замечание по поводу ввода с клавиатуры. Для ввода с клавиатуры нет встроенных файлов, поэтому вы должны пользоваться встроенными операторами.

Трудоемкий путь состоит в использовании OUT и INP ( ) для сканирования (через PPI) матрицы клавиатуры. Оператор OUT, следующий за оператором INP ( ), является одним из возможных способов знакомства каждого с аппаратурой, обеспечивающей работу клавиш.

Более очевидными являются различные варианты оператора INPUT. Имеются INPUT, LINE INPUT и функция INPUT$. Эти операторы нельзя использовать без появления курсора. Все эти операторы, за исключением INPUT$ ( ) вводят данные с эхом на экране. INPUT$ ( ) осуществляет ввод без эха, но эта функция имеет в Бейсик наивысший приоритет. INPUT будет переключать все назад от графического экрана к текстовому экрану; INPUT$ ( ) работает всегда, а LINE INPUT при работе с графическим экраном возвращает мусор.

Значительно сложнее работа с функцией INKEY$, поскольку она позволяет вам в случае необходимости собственноручно выводить на экран подсказку в виде эха введенных символов. Подобно INPUT$ (1) эта функция возвращает нулевую строку, если ввод с клавиатуры отсутствовал. Это выполняется без затруднений при любом режиме SCREEN.

4.4.2 Специфика файлового ввода-вывода

В MSX Бейсик имеется четыре блока ввода-вывода: "CRT": – вывод на экран (только в текстовом режиме); "GRP" — вывод на экран (только в графическом режиме), вы должны использовать этот вывод для посылки текста на экран при режимах SCREEN 2 или 3) ; "LPT:" стандартный вывод на печать (вы можете использовать LPRINT и LLIST, чтобы обойти его) и "CAS" — ввод-вывод для кассетного накопителя, работающего с файлами.

Для инициирования устройства и Бейсик, а также для привязки номера файла к устройству применяется оператор OPEN. Он также задает режим (подобно оператору INPUT). Для файлов имеется всего 16 номеров — от нуля до 15. Наибольший номер определяется с помощью оператора MAX FILES =, файловый номер 0 используется для операторов LOAD, CSAVE и им подобных, если это необходимо. Все файлы имеют девятибайтовый блок управления файлом (БУФ), после которого следует 256-байтный буфер для данных.

Вывод для всех устройств, работающих с файлами, осуществляется оператором PRINT (или PRINT USING). Это ваша забота – упорядочить данные так, как вы намерены их вывести; такие проблемы, как "хотел бы я вывести здесь Возврат каретки/-Перевод строки?" решаются вами. Помните: PRINT передает данные в устройство точно так же, как и на дисплей.

Имеется много способов ввода. Первый — это оператор INPUT. Он ожидает строку данных в кодах ASCII в том самом виде, в котором вы ввели ее с клавиатуры. Это означает, что вы можете разделять вводимые элементы данных запятыми, а строки заключать в кавычки.

Более удобным является оператор LINE INPUT: он считывает полную строку текста до символа RETURN (или до превышения 255-байтового предела) внутри данной строковой переменной. Вы можете использовать строковые операции, включая функцию VAL ( ), для выделения хранимых во введенной строке значений, если их больше, чем одно.

Самый прямой путь получения данных — это использование функции INPUT$ (длина, n). Эта функция возвращает заданное количество символов без какой-либо обработки.

У всех последовательных устройств в/в символ Control-Z (в коде ASCII-26 десятичное) означает конец файла. Какой-либо вывод после Control-Z неосуществим. Что же касается ввода данных, когда обнаружен знак Control-Z, то функция EOF ( ) будет указывать этот факт и данные не будут далее считываться с файла.

Для кассетных операций может использоваться необязательное имя файла: максимум длиною в шесть символов. Имеется процедура поиска файла, обеспечиваемая выражением OPEN "CAS" — смотрите операторы BLOAD и BSAVE для выяснения деталей этих поисковых процессов. Данные не загружаются в буфер при выполнении оператора OPEN: попытка ввода данных с помощью оператора INPUT первоначально будет вызывать задержку. Более того, данные, содержащиеся в буфере, при выводе не записываются до тех пор, пока он не заполнен или пока не выполняются операторы CLOSE или END — это другая задержка.

4.4.3 Загрузка и хранение программ

Для детального ознакомления с тем, что происходит при загрузке и хранении программных сегментов смотрите соответствующие части главы 3: BLOAD / BSAVE для двоичной памяти изображений, CLOAD / CSAVE – для внутриформатных программ Бейсика и LOAD / SAVE – для текстов программ.

В дополнение к этому укажем, что операции MERGE и RUN относятся к команде LOAD.

4.5 Строковые операции

Свойства системы MSX Бейсик обеспечивают богатое разнообразие строчных операций. Вы можете иметь массивы строк различной размерности. Между тем, в отличие от этого в системе Бейсик любая строка ограничена по длине 255 знаками. Для управления памятью согласованно со строковыми операциями применяются процедуры "сбора мусора".

4.5.1 Бейсики, работающие со строками.

В большинстве версий языка Бейсик фирмы Микрософт представлен один и тот же пакет для управления строками. Он имеет такие характеристики: строчная переменная, обозначаемая через DEFSTR или с помощью имени, оканчивающегося на $, имеет три байта длины каждая. Они указнвают на строковые данные длиной от 0 до 255 знаков. Если для простой константы выполняется присваивание вида a$ ="abcd", то строчная переменная в действительности будет храниться в тексте программы. Если в строке что-либо меняется, то строка перемещается в строковое пространство.

Строковое пространство является областью памяти, резервируемой для хранения строковых данных. Оператор CLEAR стирает их и определяет их длину: 200 байтов — это предел. В любой момент сформированная новая строка данных размещается на части строкового пространства. Если объем оставшегося строкового пространства недостаточен или используется функция FRE (""), то начинается "сбор мусора" и все приостанавливается до завершения.

Время "сбора мусора" является функцией размера строкового пространства и размера произвольных массивов строк. Длительность "сбора мусора" может быть до нескольких минут. "Сбор мусора" с помощью функции FRE ("") не ускоряет процесс "сбора мусора", а только несколько откладывает переполнение им строкового пространства. Однако, выбирая достаточно большое строковое пространство и накапливая до критической порции, вы можете отсрочить сбор до тех пор, пока не наберете эту критическую порцию.

Вы можете определять и использовать массивы строк точно так, как вы это делаете с числовыми массивами. Вы можете также сравнивать строки; "вес" символов определяется алфавитным порядком: "A" "B" (только латинские!!) А также "AB" "ABC", если ваши строки неравной длины. Наконец, вы можете добавить строки друг к другу с помощью операции сцепления: a$ = b$ + c$ – в a$ помещается c$, присоединенное к концу b$.

4.5.2 Строки и числа

Вы можете преобразовать любой конкретный знак в строке (от 0 до 255) в числовое значение с помощью функции ASC ( ) и преобразовать число в знак строки с помощью функции CHR$ ( ) -полезной для PRINT'a. Конечно, вы должны выбрать нужный байт в строке для обработки. Например, x= ASC ("A") передает x значение 65.

Что, если строка содержит число в формате ASCII такое, как a$ =" 345.658" ? В этом случае вы можете использовать функцию VAL ( ), чтобы преобразовать его в форму, которая позволяет использовать эту числовую величину в вашей программе. Вы также можете преобразовать любые числовые данные в строку знаков, которая будет читаться как их значение: используйте a$ = STR$ (22.6), и вы получите строку " 22.6 ", присвоенную a$.

Последний прием работает с другими числовыми базами. STR$ ( ) создает строку — в десятичной форме. Но вы можете использовать функции HEX$ ( ), OCT$ ( ) и BIN$ ( ), чтобы присвоить соответствующий числовой тип знакам строки, которые считываются так же, как эквивалент числа в знаках. Однако, вы можете использовать только целые числа.

4.5.3 Функции обработки строк

Существуют другие функции обработки строк, которые помогают обращаться со строками. Во-первых, это группа для извлечения подстрок: LEFT$ ( ), RIGHT$ ( ) и MID$ ( ). Затем существуют функции, которые порождают исходные строки в строковом пространстве: SPACE$ ( ) — чтобы задать начало чистого пространства и STRING$ ( ) — для порождения какого-нибудь знака. Вы можете определить удобную длину от 0 до 255 байтов.

Некоторые разнообразные функции: LEN ( ) — возвращает длину строки, INSTR ( ) — просматривает строку для выявления подстроки и определяет положение, если такая подстрока есть.

4.5.4 Предупредительный "сбор мусора".

Если у вас задействовало много строк или самые критические по времени операции, то накопление "мусора" может быть неприемлемо. Вы можете использовать PEEK ( ) и POKE или VPEEK ( ) и VPOKE,или SPRITE$ ( ), чтобы поместить данные на место, которое не будет использовать строковое пространство — но вы замедляете процесс и отказываетесь от всякого использования строковых переменных. Или вы можете использовать строковые операции без фактического перемещения строк.

Как это сделать? Во-первых, создайте точно достаточное строковое пространство, чтобы поместить одну копию каждой строки со (скажем) 100-байтовым остатком. Затем создайте каждую строку с помощью SPACE$ ( ). Наконец, всегда, когда вы делаете передачу с помощью строковой переменной, используйте MID$ ( ) = функцию:

MID$ (a$, 1) = строчное выражение

Это позволит избежать создания новых строк, тем не менее еще позволит использовать строковые переменные, операции и сравнения. Если вы используете какие-нибудь функции или операции, или операторы, которые заставляют использовать строковое пространство (документированные в гл. 3) и исчерпывают строковое пространство, то "сборка мусора" произойдет в любом случае.

Другой способ избежать создания новых строк — и ускорить операции – это употребить оператор SWAP. Это лучший способ для использования в любом случае — он обменивает 3— байтовые указатели и оставляет данные строки в покое.

4.6 Интерфейс с программами на машинном языке.

Совсем просто согласовать ваши процедуры на машинном языке с MSX Бейсиком. Поместите вашу машинную программу в подходящее местобуферный файл, вне пространства Бейсик или "странную переменную" (см. DEF INT и т.д.). Затем определите адрес программы и отметьте его с помощью DEFUSRn = addr. После этого сделайте вызов функции USRn ( ).

Описание USR ( ) в гл. 3 указывает форму, в которой ваша программа может получать параметры функции и возвращать результаты. Вы можете переслать (и вернуть) одну величину любого типа, какого хотите. И если настройка специального регистра не требуется, вы можете также обратиться к точкам входа базовой системы ввода-вывода (BIOS) MSX Бейсика.

4.7 Прерывания, контролируемые пользователем.

Имеется 2 способа, которыми вы можете контролировать прерывание из MSX Бейсика. Первый – это возможность прерывания 50 раз в секунду программой KEYINT IRQ через"ловушку",

расположенную в верхних адресах памяти. А также, если ваша аппаратура имеет необязательное, добавленное фирмой-изготовителем NMI — прерывание, то прерывание подобно.

Однако наиболее интересный тип организации прерываний — это прерывания из Бейсика и в Бейсике! Многие события, которые происходят "асинхронно" (в непредсказуемое время) могут быть использованы для инициации прерываний Бейсика.

5 событий прерывания — это: INTERVAL (таймер), KEY (инициированное функциональным ключом), SPRITE (столкновение, конфликтная ситуация), STOP (кнопка STOP -управления) и STRIG (кнопка джойстика или пробел клавиатуры). Каждое из них может иметь свою собственную вызывающую программу, выбранную функцией ON "событие" GOSUB: т.к. имеется 10 функциональных клавиш и 5 возможных пусковых устройств рычажкового указателя, то есть 18 различных подпрограмм прерывания, которые вы можете определить.

Имеется особый оператор, "событие" ON/OFF/STOP, который управляет прерываниями. Если это OFF, то событие игнорируется. Если это ON, то прерывание происходит. Если это STOP, то результат отмечается и копится, и прерывание произойдет, если событие ON выполнено позже. Аналогично, когда прерывание происходит для некоторого исхода, происходит автоматическое событие STOP, и возврат к событию ON, когда программа прерывания оканчивается оператором RETURN (если вы не использовали событие OFF, чтобы отменить это.) Когда прерывание принято, оно происходит после завершения выполнения текущего оператора Бейсика, и RETURN возвращает к следующему оператору.

Эта обработка результата более тщательно описана в каждом соответствующем утверждении в гл. 3.

ПРИЛОЖЕНИЯ

А. Производные функции

Функции, которые не встроены в MSX-Бейсик, могут вычисляться следующим образом.

Секанс
SEC (X) = 1/COS (X)

Косеканс
CSC (X) = 1/SIN (X)

Котангенс
COT (X) = 1/TAN (X)

Арксинус
ARCSIN (X) = ATN (X/SQR (–X * X + 1))

Арккосинус
ARCCOS (X) = –ATN (X/SQR (– X *X + 1)) + 1.5708

Арксеканс
ARCSEC (X)
= ATN (X/SQR (X*X – 1)) + SGN (SGN (X) – 1) *1.5708

Арккосеканс
ARCCSC (X)
= ATN (X/SQR (X * X – 1)) + (SGN (X) – 1) *1.5708

Арккотангенс
ARCCOT (X) = ATN (X) + 1.5708

Гиперболический синус
SINH (X) = (EXP (X) – EXP (–X)) /2

Гиперболический косинус
COSH (X) = (EXP (X) + EXP (–X)) /2

Гиперболический тангенс
TANH (X)
= (EXP (X) – EXP (–X)) / (EXP (X) + EXP (–X))

Гиперболический секанс
SECH (X) = 2/ (EXP (X) + EXP (–X))

Гиперболический косеканс
CSCH (X) = 2/ (EXP (X) – EXP (–X))

Гиперболический котангенс
COTH (X)
= (EXP (X) + EXP (–X)) / (EXP (X) – EXP (–X))

Гиперболический арксинус
ARCSINH (X) = LOG (X + SQR (X * X + 1))

Гиперболический арккосинус
ARCCOSH (X) = LOG (X + SQR (X * X – 1))

Гиперболический арктангенс
ARCTANH (X) = LOG ((1 + X) / (1 – X)) /2

Гиперболический арксеканс
ARCSECH (X) = LOG ((SQR (–X * X + 1) + 1) /X)

Гиперболический арккосеканс
ARCCSCH (X) = LOG ((SGN (X) * SQR (X * X + 1) + 1) / X)

Гиперболический арккотангенс
ARCCOTH (X) = LOG ((X + 1) / (X – 1)) /2

В. Сообщения об ошибках

Код — Сообщение

1 **NEXT without FOR**
NEXT без FOR
Переменная в операторе NEXT не соответствует переменной в операторе FOR, выполненному ранее.

2 **Syntax error**
Синтаксическая ошибка
Строка содержит неправильную последовательность символов (например, непарные скобки, неправильно написанная команда или оператор, неверная пунктуация и т.д.).

3 **RETURN without GOSUB**
RETURN без GOSUB
Для оператора RETURN не существует ранее выполнявшегося оператора GOSUB.

4 **Out of DATA**
Исчерпаны данные DATA
При выполнении оператора READ в операторе DATA не хватает данных.

5 **Illegal function call**
Неправильный вызов функции
Недопустимый параметр в математической или строковой функции. "Неправильный вызов функции" может произойти в следующих случаях:

1. Отрицательный или слишком большой индекс.
2. Отрицательный или нулевой аргумент в функциях LOG или SQR.
3. Обращение к функции USR, для которой не задан начальный адрес.
4. Использование операторов ERASE, SWAP, VARPTR с неопределенной (неиспользуемой) переменной.

5. Неправильный аргумент в операторах MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, ON ... GOTO (GOSUB) и.т.д.
6. В операторах GET или PUT записано отрицательное число.
7. В режиме SCREEN 0 или SCREEN 1 использованы графические команды/операторы.
8. Неправильные переменные в GML (графический макроязык) или в MML (макроязык музыки).
9. Использование SPRITE в режиме SCREEN 0.

6 **Overflow**
Переполнение
Результат вычисления превосходит формат представления числа в Бейсик. Если появляется UNDERFLOW (потеря значимости), то результат принимается равным нулю и выполнение программы продолжается без сообщения об ошибке.

7 **Out of memory**
Выход за пределы памяти
Программа слишком большая. Слишком много циклов FOR, GOSUB или слишком много переменных.

8 **Undefined line number**
Не определен номер строки
Обращение к несуществующей строке в операторах GOTO, GOSUB, IF ... THEN ... ELSE или DELETE.
Новая строка, которая содержит только номер строки, также вызывает это сообщение.

9 **Subscript out of range**
Выход за пределы массива
Обращение к элементу массива с индексом, превышающим размерность массива или с недопустимым количеством индексов.

10 **Redimensioned array**
Повторное задание размерности массива
Один и тот же массив упоминается в двух операторах DIM или используется оператор DIM для массива, размерность которого (10) определена ранее по умолчанию.

11 **Division by zero**
Деление на ноль
В выражении встречается деление на ноль, или возведение нуля в отрицательную степень.

12 **Illegal direct**
Неправильная команда в режиме непосредственного выполнения
В качестве команды режима непосредственного выполнения встречается оператор, недопустимый в этом режиме.

13 **Type mismatch**
Нессответствие типа
Присвоение строковой переменной числового значения или наоборот; аргументу функции, которому должно быть присвоено числовое значение, присваивается строковое значение и наоборот.

14 **Out of string space**
Исчерпано место для строковых переменных
Выход за пределы памяти, заданной в операторе для операций со строковыми переменными в Бейсик.

15 **String too long**
Слишком длинная строка
Попытка создания строки длиной более 255 символов.

16 **String formula too complex**
Строковая формула слишком сложна
Слишком длинное или слишком сложное строковое выражение. Его следует разбить на более короткие выражения.

17 **Can't continue**
Не могу продолжать
Сделана попытка продолжения программы, которая:
1. Прервана из-за ошибки.
2. Модифицирована после останова выполнения программы.
3. Не существует.

18 **Undefined user function**
Неопределенная функция пользователя
Вызов функции USR прежде чем она была определена в операторе.

19 **Device I/O error**
Ошибка устройства ввода/вывода
Ошибка в работе устройства ввода/вывода.

20 **Verify error**
Ошибка контроля
Содержимое памяти или файла при проверке оказывается другим. Эта ошибка может появиться при использовании команды CLOAD?

21 **No RESUME**
Отсутствует оператор Resume
В программе обработки ошибок нет оператора RESUME.

22 **RESUME without error**
Оператор Resume при отсутствии ошибки
Оператор RESUME встречается прежде, чем запускается программа обработки ошибок.

23 **Unprintable error**
Непечатаемая ошибка
Отсутствует сообщение об ошибке для положения, которое произошло.

24 **Missing operand**
Пропущенный операнд
Выражение содержит оператор без последующих операндов; в команде/операторе отсутствуют обязательные параметры.

25 **Line buffer overflow**
Переполнение буфера строки
Попытка ввести с помощью оператора INPUT строки, содержащей более 255 символов.

26—49 **Unprintable error**
См. 23.

50 **Field overflow**
Переполнение поля
Попытка размещения оператором FIELD количества байтов, превышающее то, которое определено для записи в произвольный файл.

51 **Internal error**
Внутренняя ошибка
В работе MSX Бейсик произошла ошибка. Для выяснения условий появления данного сообщения обращайтесь к фирме Miscrosoft.

52 **Bad file number**
Неправильный номер файла
Оператор или команда обращается к файлу с номером, для которого не был выполнен оператор OPEN, или этот номер выходит за пределы номеров, определенных при инициализации.

53 **File not found**
Файл не найден
Оператор/команда LOAD, KILL, NAME, или OPEN обращается к файлу, не существующему на данном диске.

54 **File already open**
Файл уже открыт
В режиме последовательного вывода для уже открытого файла обращение к файлу с оператором OPEN или для открытого файла происходит обращение с оператором KILL.

55 **Input past end**
Ввод после конца файла
Выполняется оператор INPUT после того, как все данные были введены в файл, или для нулевого (пустого) файла. Чтобы избежать этой ошибки, для определения конца файла используйте функцию EOF.

56 **Bad file name**
Неправильное имя файла
В операторах LOAD, SAVE, KILL или OPEN использовано неверное имя файла (например, имя файла, в котором слишком много символов).

57 **Direct statement in file**
Оператор прямого режима для файла недопустимо
При загрузке файла ASCII встретился оператор прямого режима. Выполнение команды LOAD прекращается.

58 **Sequential I/O only**
Только последовательный ввод/вывод
Использование операторов GET или PUT для файла, который был открыт (OPEN) как последовательный файл.

59 **File not OPEN**
Файл не открыт с помощью команды
Команда/оператор ввода/вывода используется для файла, который не был открыт.

60 **Bad FAT**
Неверная информация в FAT (таблице распределения файлов).
Файловая таблица распределений FAT не в порядке. Вероятно, дискета не была инициализирована с помощью команды FORMAT.

61 **Bad file mode**
Неверный режим обращения к файлу
Попытка выполнить команды PUT, GET или LOF для последовательного файла при записи (LOAD) в файл прямого доступа или выполнить оператор OPEN с неправильным режимом обращения к файлу.

62 **Bad drive name**
Неправильное имя диска
Используется неверное (незаконное) имя диска.

63 **Bad sector number**
Неправильный номер сектора
Этого случая нет в последней версии дискового Бейсик MSX.

64 **File still open**
Файл все еще открыт
Файл не закрыт командой CLOSE.

65 **File already exists**
Файл уже существует
Имя файла, определенное в операторе NAME является идентичным имени файла, уже используемому на диске.

66 **Disk full**
Диск заполнен
Использована вся дисковая память.

67 **Too many files**
Слишком много файлов
Попытка создания нового файла (использование команд SAVE или OPEN), когда все 255 элементов справочника заполнены.

68 **Disk write protected**
Защита записи диска
Диск имеет неповрежденную метку защиты записи или на него нельзя записывать.

69 **Disk I/O error**
Ошибка при вводе/выводе на диск
Неисправляемая ошибка, возникающая при выполнении операторов ввода/вывода на диск.

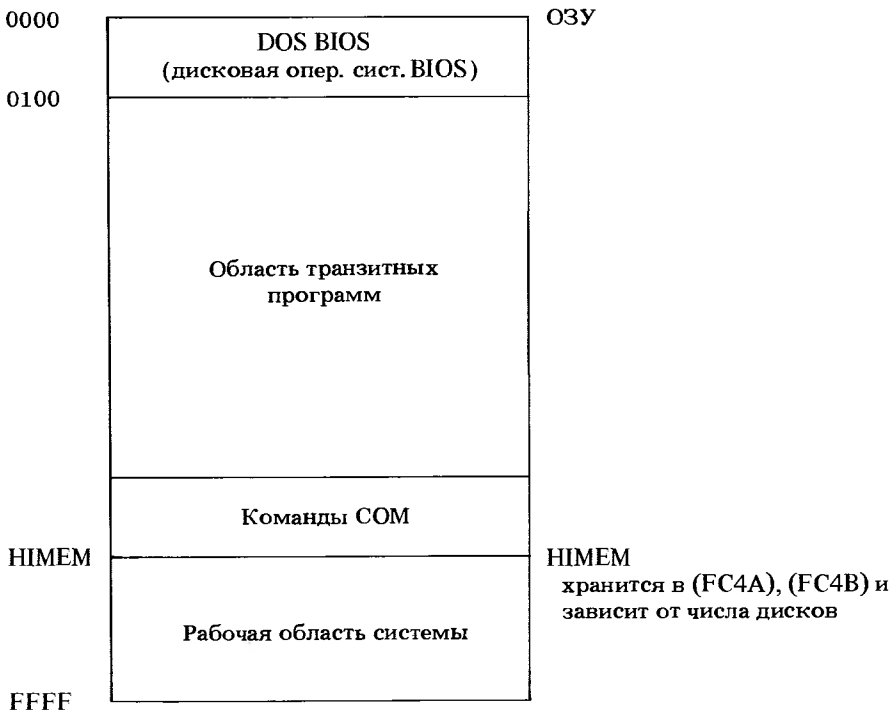
70 **Disk offline**
Дисковод в автономном режиме или выключен.

71 **Rename across disks**
Переименование с одного диска на другой
Попытка выполнения переименования файла, находящегося на другом диске. Это не разрешено.

72—255 **Unprintable error**
См. 23.

С. Карта памяти

Режим MSX-DOS (64К ОЗУ)



Режим MSX-Дисковый Бейсик (32К ОЗУ)

Режим MSX Бейсик (не дисковая версия)

D. Работа MSX-DOS

В этом разделе описываются команды операционной системы MSX-DISK. MSX-DOS обеспечивает мощные, но в то же время и простые команды, которые позволяют пользователю манипулировать информацией, хранящейся на дисках. MSX-DOS требуется компьютер MSX с 64 К ОЗУ–по крайней мере один диск и дискета, содержащая следующие файлы:

MSXDOS. SYS
COMMAND. COM

После входа в операционную систему MSX-DOS возможны два режима.

1. AUTO EXECUTION (автоматическое выполнение)

Если у вас есть файл, названный " AUTOEXEC. BAT", файл, содержащий набор команд системы MSX-DOS, этот файл будет выполняемым, как если бы команды действительно вводились с клавиатуры.

2. COMMAND MODE (командный режим).

Если автоматически выполняемый файл не существует или его выполнение закончено, система MSX-DOS перейдет в командный режим. В командном режиме вы можете или вводить команды системы MSX-DOS или просто задать имя файла с программой.

Символы маскирования

Два специальных символа, названных символами маскирования, могут быть использованы в имени файла и типе файла:

? Показывает, что любой символ может находиться в этой позиции.
* Показывает, что любой символ может занимать данную позицию или любую из оставшихся позиций в имени файла или его типе.

Символы маскирования могут быть использованы в командах COPY, DEL, DIR, REN, или TYPE.

Краткое описание команд MSX-DOS

BASIC

Цель: войти в режим DISK BASIC

Формат: BASIC [имя файла]

Действие: Эта команда переведет вас в режим DISK BASIC Дополнительно можно указать имя файла программы, написанной на BASIC.

COPY

Цель: копирование файлов с одной дискеты на другую.

Формат: COPY файл-источник [/ключ] файл-приемник [/ключ]

Действие: Эта команда копирует один или более файлов с одной дискеты на другую. Файл-источник может быть одним или списком файлов, объединенных знаком +. Эта функция позволяет объединять список текстовых файлов в один. Если файл-приемник отсутствует, используется первое имя в файле-источнике. Значения ключей следующие:

/A Указывает, что файл относится к ASCII файлу.
/B Указывает, что файл относится к BINARY (двоичному) файлу.
/V Проверяет копируемый файл.

DATE

Цель: Установка или печать системной даты

Формат: DATE

Действие: После введения команды DATE система выдаст следующее:

Текущая дата: месяц-число-год.
Введите новую дату.

Если вы не хотите изменять системную дату, просто нажмите клавишу RETURN. В противном случае введите дату.

Область задания года 1980-2099.

Область задания месяца 1-12.
Область задания числа 1-31.
День недели автоматически вычисляется в системе MSX-DOS.

Если вы не введете дату, в системе будет использована дата по умолчанию 01-01-1984.

DEL

Цель: Уничтожение файла из каталога.

Формат: DEL имя-файла

Действие: При использовании команды DEL названный файл будет уничтожен из каталога. Если вы напишете DEL *.*, компьютер будет спрашивать "Are you sure?": ("Вы уверены?"). Если вы ответите "y" (да), все файлы на дискете будут уничтожены.

DIR

Цель: Распечатка каталога диска (drive).

Формат: DIR [имя-файла] [/ключ]

Действие: Эта команда позволяет просмотреть каталог диска. Если "имя-файла" указано, только зти файлы будут показаны. Значения "ключа" следующие:

/P Остановка при заполнении экрана перечнем каталога
/W Печать только имен файлов (если/W не используется, команда DIR печатает также размер и дату создания файла).

FORMAT

Цель: Форматирование дискеты.

Формат: FORMAT

Действие: Эта команда разрушает любую информацию, хранимую на дискете, и создает область для использования. Все новые дискеты должны быть отформатированы перед использованием. Когда вы используете эту команду, компьютер будет спрашивать:

"Format which drive (A or B) ?"
(Форматирование какого дисковода (A или B) ?).

Вы должны ответить A или B.

MODE

Цель: Выбор размера экрана дисплея

Формат: MODE размер

Действие: Если "размер" находится в интервале 1—32, используется режим SCREEN 1 (экран 1).

Если размер находится в интервале 33—40, используется режим SCREEN 0 (экран 0).

PAUSE

Цель: Задерживается выполнение до нажатия клавиши.

Формат: PAUSE

Действие: При использовании этой команды система выдает сообщение "Strike a key when ready". ("Нажмите клавишу, если вы готовы ...").

Эта команда полезна при нахождении ее в файле AUTOEXEC. BAT.

REM

Цель: Запись строк комментарий.

Формат: REM [комментарии]

Действие: Компьютер не интерпретирует и ничего не делает над командной строкой, следующей за командой REM. Используется только в файле AUTOEXEC. BAT для комментирования.

REN

Цель: Переименование файлов.

Формат: REN стар-имя-файла нов-имя-файла

Действие: Эта команда используется для изменения имен существующих файлов.
Если стар-имя-файла не существует или нов-имя-файла уже существует, будет выдаваться ошибка.

TIME

Цель: Установка или печать системного времени.

Формат: TIME час : минута : секунда

Действие: Команда TIME устанавливает время на системных часах для использования его во временных отметках при создании файлов в каталоге.
час 0—23 ::= час дня, используется 24 часа
минута 0—59 ::= минуты в пределах газа
секунда 0—59 ::= секунды

TYPE

Цель: Распечатка текста файла на экране

Формат: TYPE имя-файла

Действие: Команда TYPE читает файл и выводит его содержимое на экран.

E. MSX–DISK BASIC

В этом разделе описываются команды системы MSX-DISK BASIC Эти команды могут использоваться в компьютерах MSX с памятью (RAM) размером 16 К, 32 К или 64 К при наличии по крайней мере одного диска. Система MSX-DISK BASIC начинает работать, если система MSX-DOS не может быть вызвана или если введена команда "BASIC" в командном режиме системы MSX-DOS

Если системы MSX-DOS нет в наличии на вашей машине, система MSX-DISK BASIC может быть включена в одном из следующих случаев:

1. AUTO EXECUTION (автоматическое выполнение)

Если вы имеете файл, названный "AUTOEXEC. BAS" файл, который содержит BASIC-программу, эта BASIC-программа является выполняемой.

2. COMMAND MODE (командный режим).

Если автоматически выполняемый файл не существует, вы запросите ввод даты, и система войдет в командный режим системы DISK BASIC.

Перечень команд и функций системы MSX-DISK BASIC

BLOAD

Цель: Запись файла в двоичном формате.

Формат: BLOAD имя-файла [,R][,смещение]
,S

Действие: Эта команда загружает указанный файл в память. Если указано "смещение", загрузка файла производится по адресу, равному сумме "смещения" и начального адреса файла.
При флаге R будет автоматически выполняться загруженная в память программа.
При флаге S будет загружаться файл в память VRAM без выбора экранного режима (screen mode).

BSAVE

Цель: Считывание файла в двоичном формате

Формат: BSAVE имя-файла , start , end [, выполнение]
S

Действие: Эта команда будет считывать информацию, которая располагается в памяти между двумя адресами, указанными в "start" и "end". Если адрес "выполнение" опущен, будет использоваться адрес "start". Если символ "S" указан, данные будут считываться в память VRAM.

CLOSE

Цель: Закрытие файлов

Формат: CLOSE [[#] номер-файла [,[#] номер-файла]...]

Действие: Если номер-файла опущен, все открытые файлы будут закрыты. В противном случае будут закрываться только указанные в команде файлы.

COPY

Цель: Копирование файлов.

Формат: COPY имя-файла-источника [TO имя-файла-приемника]

Действие: Эта команда будет копировать файлы, имеющие любой формат. Если слова "TO имя-файла-приемника" опущены, файл источник будет скопирован на активизированный диск.

CVI,CVS,CVD

Цель: Преобразование символьной величины в числовую величину

Формат: CVI (2-байтная строка)
CVS (4-байтная строка)
CVD (8-байтная строка)

Действие: В произвольном дисковом файле числовая величина хранится как строковая:

целая : : 2-байтная строка
одинарная точность : : 4-байтная строка
двойная точность : : 8-байтная строка
Эта функция преобразует эти строковые величины в числовые.

DSKF

Цель: Определение свободного пространства на гибком диске

Формат: DSKF (номер-диска)

Действие: Эта функция дает число оставшихся блоков на гибком диске, указанном в команде. Номер-диска соответствует действительному имени диска таким образом:
0 — активизированный диск
1 — диск A
2 — диск B

EOF

Цель: Контролирует достижение конца файла

Формат: EOF (номер-файла)

Действие: Эта функция возращает -1 (истина), если конец указанного файла достигнут. В противном случае-возврат 0 (ложь)

FIELD

Цель: Назначение буферного поля для переменных

Формат: FIELD [#] файл No., поле 1 AS строк-пер 1[, поле 2 AS строк-пер 2] ....

Действие: Эта команда должна быть выполнена прежде, чем вы обратитесь к файлу прямого доступа при использовании команд PUT и GET "поле N" назначает количество символов, которое может быть определено в "строк-пер N" (строковой переменной N), Разрешается использовать несколько команд FIELD с различным форматом для одного и того же буфера.

FILES

Цель: Печать имен файлов на диске

Формат: [L] FILES [имя-файла]

Действие: Если имя-файла указано в команде, имя файла будет распечатано при условии нахождения файла на диске. Если имя-файла отсутствует, все файловые имена, которые существуют на диске, будут распечатаны. Имя-файла может включать символы "?" и "*" как и в системе MSX-DOS. Команда LFILES производит печать на принтере.

FORMAT

Цель: Форматирование гибкого диска

Формат: CALL FORMAT

Действие: Эта команда будет форматировать гибкий диск для пользования в системах MSX-DISK BASIC или MSX-DOS. Смотрите команду FORMAT в разделе команд системы MSX-DOS.

GET

Цель: Чтение записи из файла прямого доступа на диске

Формат: GET [#] номер-файла [, номер-записи]

Действие: Используя эту команду, вы сможете читать запись из файла на диске в буфер файла прямого доступа. Если номер-записи отсутствует, будет считаться следующая запись (после последней в GET).

INPUT

Цель: Чтение данных из последовательного файла на диске

Формат: INPUT # номер-файла, список-перем

Действие: Эта команда используется для чтения элементов данных из последовательного файла на диске и присвоения их переменным.

INPUT$

Цель: Получение строки заданной длины, чтение ее из файла на диске

Формат: INPUT$ (длина, [#] номер-файла)

Действие: Эта функция возвращает строку символов длиной "длина", чтение из файла, заданного номером-файла.

KILL

Цель: Удаление файла с диска

Формат: KILL имя-файла

Действие: Эта команда используется для удаления файлов с диска. Может быть удален файл любого типа. Однако, если обозначенный файл является открытым, выдается сообщение об ошибке "File already open" (файл все еще открыт").

LINE INPUT

Цель: Чтение целой строки из последовательного файла на диске в символьную переменную.

Формат: LINE INPUT # номер-файла, симв.-перем.

Действие: Эта команда читает целую строку, заканчивающуюся кодами CR и LF, из последовательного дискового файла и присваивает значение этой строки символьной переменной. Строка может доходить до 256 символов, включая CR и LF.

LOAD

Цель: Загрузка программы, написанной на BASIC, в память

Формат: LOAD имя-файла [, R]

Действие: Эта команда используется для загрузки BASIC – программы (в двоичном формате или формате ASCII) с диска в память. Если имеется флаг R, программа будет выполняться (RUN) после ее загрузки.

LOC

Цель: Получение номера текущей записи

Формат: LOC (номер-файла)

Действие: Команда LOC возвращает номер записи из произвольных дисковых файлов, только что прочитанных или записанных с помощью команд GET или PUT. Для последовательных файлов команда LOC возвращает номер секторов (128 байт в блоке). считанных или записанных в файл с момента, когда он был открыт.

LOF

Цель: Получение длины файла

Формат: LOF (номер-файла)

Действие: Эта функция возвращает длину указанного в команде файла в байтах.

LSET,RSET

Цель: Перемещение данных из памяти в буфер файлов прямого доступа

Формат: LSET смив-перем. = симв.-выраж.
RSET симв-перем. = симв.-выраж.

Действие: Если "симв. выраж." требуется меньше байтов, чем их имеется в поле символьной переменной, команда LSET выравнивает строку по левому краю, а команда RSET— по правому краю, вставив дополнительные пробелы.

MERGE

Цель: Объединение указанного файла на диске с программой в памяти

Формат: MERGE имя-файла

Действие: Эта команда используется для объединения двух программ. Программный файл, который должен быть объединен, может храниться в формате ASCII. Если некоторые строки совпадают, строки из файла на диске заместят соответствующие строки в памяти.

MKI$,MKS$,MKD$

Цель: Преобразование числовой величины в символьную величину

Формат: MKI$ (целочисл.-выраж.)
MKS$ (выраж.-с одинар.-точностью)
MKD$ (выраж.-с двойной точностью)

Действие: Некоторые числовые величины, которые расположены в произвольном буфере файлов с помощью команд LSET или RSET, могут быть преобразованы в символьные величины, используя данные функции.

NAME

Цель: Изменение имени файла на диске

Формат: NAME стар.-имя-файла AS новое-имя-файла

Действие: "Стар.-имя-файла" должно существовать, "новое-имя-файла" можно не указывать; в противном случае выдается ошибка.

OPEN

Цель: Назначение буфера файла для ввода/вывода (I/O)

Формат: OPEN имя-файла [FOR режим] AS [#] номер-файла [LEN = длина-записи]

Действие: Эта команда назначает буфер файла для ввода/вывода и определяет режим ввода/вывода. Имеется 4 разных режима доступа:

1. FOR INPUT определяет последовательный ввод
2. FOR OUTPUT определяет последовательный вывод
3. FOR APPEND определяет последовательный вывод с конкатенацией

4. Отсутствие режима — определяет режим прямого доступа номер-файла может достигать 15, не превышая предельное значение, указанное командой MAXFILES.

"длина-записи" устанавливает наибольшую длину записи (1-256) для файла прямого доступа. Если этот параметр отсутствует, длина записи будет равна 256.

PRINT #, PRINT # USING

Цель: Вывод данных в последовательный файл данных

Формат: PRINT # номер-файла, [USING симв.-выраж.;] [выраж.] ...

Действие: Эти операторы выполняются точно так же как операторы PRINT и PRINT USING, за исключением того, что вывод производится в файл, вместо вывода на экран.

PUT

Цель: Перемещение записи из буфера файла прямого доступа в файл.

Формат: PUT [#] номер-файла [,номер-записи]

Действие: Этот оператор переписывает содержимое буфера в указанный файл. номер-записи может принимать значения от 1 до 4294967295. Если этот параметр отсутствует, будет переписана следующая запись (после последней команды PUT/GET).

RUN

Цель:: Загрузка программного файла с диска и запуск этой программы на выполнение

Формат: RUN {имя-файла [,R]}
[номер-строки]

Действие: Если "имя-файла" не указано, команда RUN ничего не выполняет с дисками. Команда RUN загружает файл с именем "имя-файла" и выполняет программу, установив в исходное состояние переменные, и закрыв все открытые файлы. Однако, если флаг R указан, все файлы данных остаются открытыми.

SAVE

Цель: Сохранение программы на языке BASIC, находящейся в памяти, в виде файла на диске.

Формат: SAVE имя-файла [,A]

Действие: Эта команда сохраняет программу в виде файла на диске. Если на диске уже есть файл с "именем-файла", старый файл будет заменен на новый. При использовании флага A файл будет сохранен на диске в формате ASCII.

SYSTEM

Цель: Передается управление на систему MSX-DOS

Формат: CALL SYSTEM

Действие: Эта команда находит применение только, если ранее система MSX-DISK BASIC была вызвана из командного режима системы MSX-DOS. Перед возвратом в систему MSX-DOS все открытые файлы будут закрыты, а программа и данные, находящиеся в памяти, будут потеряны.

VARPTR

Цель: Определение адреса переменной или адреса буфера

Формат: VARPTR {(имя-переменной)}
[#] номер-файла

Действие: Если обозначено "имя-переменной", будет возвращен адрес этой переменной. Если обозначен "номер-файла", будет возвращен начальный адрес блока управления файлом (FCB).

F. Аппаратные средства

LSIs	**(БИСы и память)**	
CPU	**(центральный процессор)**	Z-80A совместимость 3.579545 MHz 1 цикл ожидания на M1 цикл
VDP	**(видео дисплей процессор)**	TMS–9918A совместимость
PSG	**(программируемый генератор звука)**	AY–3–8910 совместимость
PPI	**(программируемый периферийный интерфейс)**	1–8255 совместимость
ROM	**(постоянная память ПЗУ)**	интерпретатор MSX Бейсик 32К байт
RAM	**(ОЗУ, ЗУ)**	16К, 32К или 64К байт

Таблица ввода/вывода (I/O map)

ADRS	R/W	PURPOSE	
(адрес)	(Чтение/запись)	(назначение)	
(90)	W	Строб OUT (B0) длиной OUT (вывод)	
	R	Статус IN (B1) 1= BUSY (занято)	
(91)	W	Печать данных длиной OUT (вывод)	
(98)	W	Запись данных в память VRAM	VDP
	R	Чтение данных из памяти VRAM	
(99)	W	Установка команды, адреса	
	R	Чтение статуса	
(A0)	W	Длина адреса	PSG
(A1)	W	Запись данных	
(A2)	R	Чтение данных	
(A8)	W	Запись данных в порт A	PPI
	R	Чтение данных из порта A	
(A9)	W	Запись данных в порт B	
	R	Чтение данных из порта B	
(AA)	W	Запись данных в порт C	
	R	Чтение данных из порта C	
(AB)	W	Установка режима	

PPI PORTS (порты программируемого периферийного интерфейса)

PORT (порт)	BIT (бит)	I/O (ввод/вывод)	NAME (имя)	MEANING (Значение)
1	2	3	4	5
A	0 1	OUTPUT (ВЫВОД)	CS0L CS0H	выбор установочного места для 0000—3FFF
	2 3		CS1L CS1H	выбор установочного места для 4000—7FFF
	4 5		CS2L CS2H	выбор установленного места для 8000—BFFF
	6 7		CS3L CS3H	выбор установленного места для C000—FFFF
B	0 1 7	INPUT (ввод)		состояние строки клавиатуры
C	0 1 2 3	OUTPUT (ВЫВОД)	KB0 KB1 KB2 KB3	выбор строки при сканировании клавиатуры
	4		CASON	CAS CTRL (L–ON) кассетник включен мотор
	5		CASW	запись на кассету
	6		CAPS	CAPS LAMP (L–ON) Маленькие буквы CAPITALS большие
	7		SOUND (звук)	вывод звука (SOFT)

→ Управление лампочками на клавиатуре

СОЕДИНЕНИЯ

1. DIN – 5 контактный разъем

PIN (контакт)	SIGNAL NAME (имя сигнала)
1	+ 5V
2	GND (земля)
3	AUDIO
4	MONITOR VIDEO
5	RF VIDEO

2. Разъем на кассете

PIN (контакт)	SIGNAL (сигнал)
1	GND (земля)
2	GND (земля)
3	GND (земля)
4	CMTOUT
5	CMTIN
6	REM +
7	REM –
8	GND (земля)

3. Разъем координатной ручки (joystick)

PIN (контакт)	SIGNAL (сигнал)
1	FWD (вперед)
2	BACK (обратно)
3	LEFT (влево)
4	RIGHT (вправо)
5	+ 5V
6	TRG1 (триггер 1)
7	TRG2 (триггер 2)
8	OUTPUT (выход)
9	GND (земля)

4. Сигналы шины

PIN	NAME	I/O	PIN	NAME	I/O
контакт	имя	ввод/вывод	контакт	имя	ввод/вывод
1	CS1	O	2	CS2	O
3	CS12	O	4	SLTSL	O
5	–	–	6	RFSH	O
7	WAIT (задержка)	I	8	INT	I
9	M1	O	10	BUSDIR	I
11	IORQ	O	12	MERQ	O
13	WR	O	14	RD	O
15	RESET(сброс)	O	16	–	–
17	A9	O	18	A15	O
19	A11	O	20	A10	O
21	A7	O	22	A6	O
23	A12	O	24	A8	O
25	A14	O	26	A13	O
27	A1	O	28	A0	O
29	A3	O	30	A2	O
31	A5	O	32	A4	O
33	D1	I/O	34	D0	I/O
35	D3	I/O	36	D2	I/O
37	D5	I/O	38	D4	I/O
39	D7	I/O	40	D6	I/O
41	GND (земля)	–	42	CLOCK (тактовый генератор)	O
43	GND (земля)	–	44	SW1	–
45	+5V	–	46	SW2	–
47	+5V	–	48	+12V	–
49	SUNDIN (ввод звука)	I	50	–12V	–

G. Список входных точек BIOS

ENTRY (входная величина) — **FUNCTION** Функция

003E — **Инициализация функциональных клавиш**
MODIFY (изменяются все регистры)

004A — **Чтение данных из памяти VRAM**
IN (вход) HL : адрес VRAM
OUT (выход) A : данные
MODIFY (изменяются) AF (регистры A и F)

004D — **Запись данных в память VRAM**
IN (вход) HL : адрес VRAM
A : данные
MODIFY (изменяются) AF (регистры A и F)

0056 — **Заполнение память VRAM константами**
IN (вход) BC : длина
HL : VRAM адрес
A : данные
MODIFY AF, BC (изменяются регистры AF, BC)

0059 — **Пересылка блока данных из памяти VRAM в указанную память**
IN (вход) BC : длина
DE : адрес приемника памяти RAM
HL : адрес источника памяти VRAM
MODIFY (изменяются все регистры)

005C **Пересылка блока данных из указанной памяти в память VRAM**
IN (вход) BC : длина
DE : адрес приемника памяти VRAM
HL : адрес источника памяти RAM
MODIFY (изменяются все регистры)

0090 **Инициализация PSG (Программируемого звукового генератора)**
MODIFY (изменяются все регистры)

0093 **Запись данных в PSG**
IN (вход) A : номер регистра
E : данные

0096 **Чтение данных из PSG**
IN (вход) A : номер регистра
OUT (выход) E : данные
MODIFY (изменяется регистр A)

009C **Проверка буфера клавиатуры**
OUT (выход) Ноль (флаг), если буфер пустой
MODIFY (изменяются регистры AF)

009F **Ожидание ввода клавиши (INPUT)**
OUT (выход) A : символ
MODIFY (изменяются регистры AF)

00D5 **Проверка состояния координатной ручки (joystick)**
IN (вход) A : номер ручки ID (0–2)
OUT (выход) A : состояние ручки (0–8)
MODIFY (изменяются все регистры)

00D8 **Проверка кнопки**
IN (вход) A : номер триггера ID (0–4)
OUT (выход) A : =255, если нажата
MODIFY (изменяются регистры AF)

0141 **Получение состояния матрицы клавиатуры**
IN (вход) A : адрес строки матрицы
OUT (выход) A : состояние строки
MODIFY (изменяются регистры AF)

0156 **Сброс буфера клавиатуры**
MODIFY (изменяются регистры HL)

Н. Системные переменные

Адрес	Содержание
F380	программа чтения из главного установочного места
F385	программа записи в главное установочное место
F38C	переход на программу записи в главное установочное место
F39A	начальные адреса для VSR0–9
F3AE	длина строки = 39
F3AF	длина строки = 31
F3B0	длина строки
F3B1	количество строк на экране = 24
F3B2	интервал между столбцами = 14
F3B3	SCREEN 0 (экран 0) имя таблицы
F3B5	цветная таблица
F3B7	форма символа
F3B9	характеристика
F3BB	спрайт
F3BD	экран 1 имя таблицы
F3BF	цветная таблица
F3C1	форма символа
F3C3	характеристика
F3C5	спрайт
F3C7	экран 2 имя таблицы
F3C9	цветная таблица
F3CB	форма символа
F3CD	характеристика
F3CF	спрайт
F3D1	экран 3 имя таблицы
F3D3	цветная таблица
F3D5	форма символа
F3D7	характеристика
F3D9	спрайт
F3DB	ключ клик
F3DC	курсор Y
F3DD	курсор X
F3DE	функциональные клавиши
F3DF	VDP содержимое регистра
F3E7	= 0
F3E8	= (FF)

F3E9	высокоприоритетный (основной) цвет
F3EA	фоновый цвет
F3EB	граничный цвет
F3EC	переход 0
F3EF	переход 0
F3F2	характеристика байта
F3F3	адрес таблицы очередей
F3F5	= (FF)
F3F6	ключ развертки синхронизации
F3F7	= 50
F3F8	заполнение ключевого буфера
F3FA	выбор из ключевого буфера
F3FC	параметры ввода/вывода кассеты
F40F	указатель оператора RESUME NEXT
F414	кодовая ошибка
F415	печатающая головка
F416	вывод принтера
F417	= 0 для принтера MSX
F418	не ноль, если выводится необработанный символ
F419	значимая функция
F41C	курсор строки
F41F	скоростной буфер
F55D	запятая для INPUT (команды ввода)
F55E	буфер ввода
F660	конец буфера
F661	позиция на терминале
F662	флаг массива
F663	значение типа
F664	тип оператора
F665	для скоростной работы
F666	указатель текста для выбора символа
F668	внутренняя форма константы после выбора символа
F669	тип константы
F66A	значение константы
F672	вершина памяти
F674	возможная вершина стека
F676	вершина текста
F678	временный описатель
F67A	хранение временных описателей
F698	строка описателя после последовательности операторов
F69B	возможная вершина строки
F69D	по сбору мусора
F6A1	указатель оператора FOR

F6A3	указатель оператора DATA
F6A5	рабочий флаг для FOR и USR
F6A6	рабочий флаг для INPUT и READ
F6A7	работа операторов
F6A9	= 0, если не программная строка
F6AA	= 0, если режим AUTO (автоматический)
F6AB	строковое число AUTO
F6AD	инкремант AUTO
F6AF	указатель текста для оператора RESUME
F6B1	сохранить стек для обработки ошибок
F6B3	строковая ошибка
F6B5	текущая строка
F6B7	указатель текста для RESUME
F6B9	строка обработки ошибки
F6BB	= 1, если действительны обрабатываемые ошибки
F6BC	временная работа
F6BE	старый номер строки, установленный для CTRL-STOP, STOP, END
F6C0	старый текстовый указатель
F6C2	начальный адрес простых переменных
F6C4	начальный адрес массивов
F6C6	конец пользовательской памяти
F6C8	указатель для оператора DATA
F6CA	тип переменной для A–Z
F6E4	стек для сбора мусора
F6E6	длина таблицы
F6E8	таблицы параметров для функций, определяемых пользователем
F74C	указатель блока параметров
F74E	длина блока параметров
F750	адреса параметров
F7B4	флат источника параметров
F7B5	конец источника
F7B7	= 0, если не подходящая функция
F7B8	временная работа по сбору мусора
F7BA	количество назначаемых функций
F7BC	работа по обмену
F7C4	= 0, если выключено трассирование
F7C5	рабочая область для пакета программ BCD
F85F	область данных для обрабатывающего файла
F87F	состояние функциональных ключей
F91F	таблицы VRAM BASE
F92A	для GENGRP

F931	рабочая область для оператора CIRCLE
F949	рабочая область для оператора PAINT
F956	рабочая область для оператора PLAY
FBB0	возможность "теплого" старта, если не ноль
FBB1	не ноль, если текст на Бейсике находится в памяти ROM
FBB2	ограничительная строка таблицы
FBCA	позиция первого символа в INLIN
FBCC	код кусора
FBCD	флаг для функциональных ключей
FBCE	флаги случайных ловушек для функциональных ключей
FBD8	случайный флаг
FBD9	флаг клик
FBDA	статус старого ключа
FBE5	статус нового ключа
FBF0	ключ буфера кодов
FC18	работа экранного обработника
FC40	работа о преобразователя образца
FC48	нижняя граница памяти
FC4A	верхушка памяти
FC4C	таблица ловушек
FC9A	RTYCNT
FC9B	INTFLG
FC9C	PAD Y
FC9D	PAD X
FC9E	JIFFY
FCA0	интервал
FCA2	счетчик интервалов
FCA4	чтение с кассеты
FCA6	заголовок графического символа
FCA7	последовательность переключения счетчика
FCA8	вставляемый флаг
FCA9	курсор ON/OFF
FCAA	курсор символа
FCAB	статус ключа CAPS
FCAC	заблокированный рабочий ключ
FCAD	не используется
FCAE	= 0, пока загружается программа на Бейсике
FCAF	режим экрана
FCB0	старый режим экрана
FCB1	символ для CAS:
FCB2	граничный цвет для PAINT
FCB3	графический курсор X
FCB5	графический курсор Y

FCB7	графический аккумулятор X
FCB9	графический аккумулятор Y
FCBC	масштаб DRAW
FCBD	угол DRAW
FCBE	BLOAD/BSAVE
FCBF	старт BSAVE
FCC1	рабочая область установочного места
FD9A	ловушки

I. Зарезервированные слова

Следующие слова системы MSX-Бейсик зарезервированы как ключевые слова, которые не могут быть использованы как часть переменной.

ABS	(эй би эс)	DEFSNG	(деф-эс-эн-джи)
AND	(энд)	DEFSTR	(деф-эс-ти-ар)
ASC	(эй эс си)	DELETE	(де-лит)
ATN	(эй ти эн)	DIM	(дим)
ATTR$	(эй ти ти-а)	DRAW	(дро)
AUTO	(ото)	DSKF	(ди-эс-кэй-эф)
BASE	(бейз)	DSKI$	(ди-эс-кей-ай)
BEEP	(бип)	DSKO$	(ди-ско)
BIN$	(би-ай-эн)	ELSE	(елз)
BLOAD	(би-доуд)	END	(енд)
BSAVE	(би сейв)	EOF	(и-оу-эф)
CALL	(кол)	EQV	(и-кю-веу)
CDBL	(си-ди-би-эл)	ERASE	(э-рэйз)
CHR$	(си-эйч а)	ERL	(и-а-эл)
CINT	(си-инт)	ERR	(е-а-а)
CIRCLE	(сёкл)	ERROR	(еррор)
CLEAR	(клиар)	EXP	(е-кс-п)
CLOAD	(силоуд)	FIELD	(филд)
CLOSE	(клоуз)	FILES	(файлз)
CLS	(си-эл-эс)	FIX	(фикс)
CMD	(си-эм-ди)	FN	(эф-эн)
COLOR	(калор)	FOR	(фор)
CONT	(конт)	FPOS	(эф-пос)
COPY	(копи)	FRE	(эф-эр-и)
COS	(си-оу-эс)	GET	(гет)
CSAVE	(си-сейв)	GO TO	(гоу-ту)
CSNG	(си-эс-эн-джи)	GOSUB	(гоу-саб)
CSRLIN	(си-эс-эр-лин)	GOTO	(гоу-ту)
CVD	(си-ви-ди)	HEX$	(хекс)
CVI	(си-ви-ай)	IF	(иф)
CVS	(си-ви-эс)	IMP	(и-эм-пи)
DATA	(дейта)	INKEY$	(ин-кей)
DEF	(ди-и-эф)	INP	(и-эн-пи)
DEFDBL	(деф-ди-би-эл)	INPUT	(ин-пут)
DEFINT	(деф-инт)	INSTR	(ин-эс-ти-эр)

INT	(инт)
IPL	(ай-пи-эл)
KEY	(кей)
KILL	(кил)
LEFT$	(лэфт)
LEN	(эл-и-эн)
LET	(лэт)
LFILES	(эл-файлз)
LINE	(лайн)
LIST	(лист)
LLIST	(эл-лист)
LOAD	(лоуд)
LOC	(лок)
LOCATE	(лоу-кейт)
LOF	(лоф)
LOG	(лог)
LPOS	(эл-пос)
LPRINT	(эл-принт)
LSET	(эл-сет)
MAX	(макс)
MERGE	(мёлж)
MID$	(эм-ай-ди)
MKD$	(эм-ка-де)
MKI$	(эм-ка-и)
MKS$	(эм-ка-эс)
MOD	(мод)
MOTOR	(моу-тор)
NAME	(нэйм)
NEW	(нью)
NEXT	(некст)
NOT	(нет)
OCT$	(окт)
OFF	(оф)
ON	(он)
OPEN	(оу-пен)
OR	(ор)
OUT	(аут)
PAD	(пэд)
PAINT	(пэйнт)
PDL	(пи-дэ-эл)
PEEK	(пик)
PLAY	(плей)
POINT	(поинт)
POKE	(поук)
POS	(пос)
PRESET	(пи-ар-и-сет)
PRINT	(принт)
PSET	(пи-сет)
PUT	(пут)
READ	(рид)
REM	(рем)
RENUM	(ренум)
RESTORE	(рестор)
RESUME	(рез-юм)
RETURN	(ретён)
RIGHT$	(райт)
RND	(эр-эн-ди)
RSET	(эр-сет)
RUN	(ран)
SAVE	(сэйв)
SCREEN	(скрин)
SET	(сет)
SGN	(эс-джи-эн)
SIN	(син)
SOUND	(саунд)
SPACE$	(спейс)
SPC (	(эс-пи-си)
SPRITE	(спрайт)
SQR	(эс-ку-эр)
STEP	(степ)
STICK	(стик)
STOP	(стоп)
STR$	(эс-тэ-эр)
STRIG	(стриг)
STRING$	(стринг)
SWAP	(своп)
TAB (	(таб)
TAN	(тан)
THEN	(зен)
TIME	(тайм)
TO	(ту)
TROFF	(ти-роф)
TRON	(ти-рон)
USING	(юзинг)
USR	(ю-эс-эр)
VAL	(вэл)

VARPTR	(вар-пэ-тэ-эр)
VDP	(вэ-дэ-пэ)
VPEEK	(вэ-пик)
VPOKE	(вэ-поук)
WAIT	(вэйт)
WIDTH	(видз)
XOR	(экс-ор)

ИНДЕКС

T

U

V

W

MEMO

MEMO

MEMO

http://www.gr8bit.ru