Ваш компьютер - в Ваших руках!

ZX SPECTRUM "ИЗНУТРИ"

защита и взлом программ

48k Sinclair

АННОТАЦИЯ

Мало кому нравятся программы, которые при первой же ошибке в ответе или при нахатии клавиши "BREAK" очищают всю память компьютера, не оставляя после себя никакого следа, либо зависают, вынуждая нахимать "RESET". Ситуация перестает быть забавной, когда мы имеем какую-нибудь будь программу, которую хотим приспособить для нетипового оборудования (например, джойстика) или хотим заменить в программе все английские тексты на русские, а программу не удается остановить.

В этом описании будет последовательно дана информация, необходимая при такого рода работе на Вашем компьютере: карта памяти, способ записи в памяти отдельных строк БЕЙСИКа, важные системные переменные и т.п. Позже мы доберемся до считывания программ и блоков данных с ленты "безопасным" способом, т.е. так, чтобы они не стартовали автоматически и можно было бы рассмотреть их содержимое. В конце мы займемся также обезвреживанием предохранителей, написанных на внутреннем языке. Постараемся все это иллюстрировать конкретными примерами на известных программах. Надеемся, что наши усилия не пропадут зря и Вы тоже научитесь добираться до каждой программы.

1. РАСПРЕДЕЛЕНИЕ ПАМЯТИ

В принципе память поделена на две основные части: ROM и RAM. ROM занимает адреса 0..16383, RAM - адреса 16384..65535. Содержимым ROM мы сейчас заниматься не будем, но зато внимательно присмотримся к памяти RAM. Она поделена на блоки, выполняющие различные функции в системе БЕЙСИКа (рис.1).

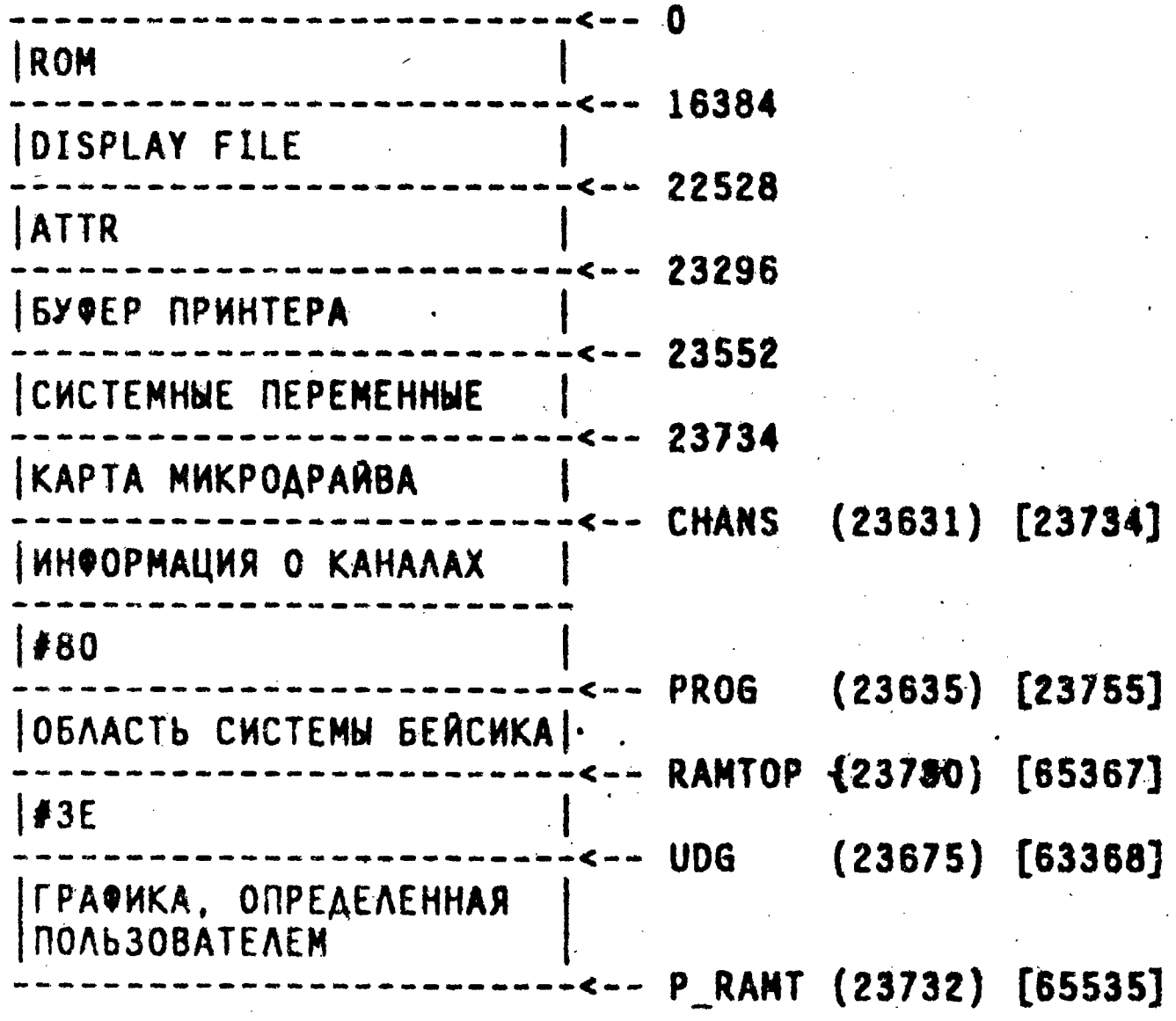

Рис.1. Распределение памяти

DISPLAY FILE - Область, в которой хранится информация о том,

включены или погашены последовательный точки экрана. Это занимает 6144 байта.

ATTR - Эти 768 байт (адреса 22528) определяют цвета последовательных полей экрана (8*8 точек).

БУФЕР ПРИНТЕРА - Эта область (от 23296 до 23551) используется лишь во время работы компьютера с принтером. Если не используются инструкции, касающиеся принтера (LLIST, LPRINT, COPY), то его содержимое не меняется и его можно использовать для других целей. Помните только, что использование какой-либо из этих инструкций даже без подключения принтера произведет в этой области изменения.

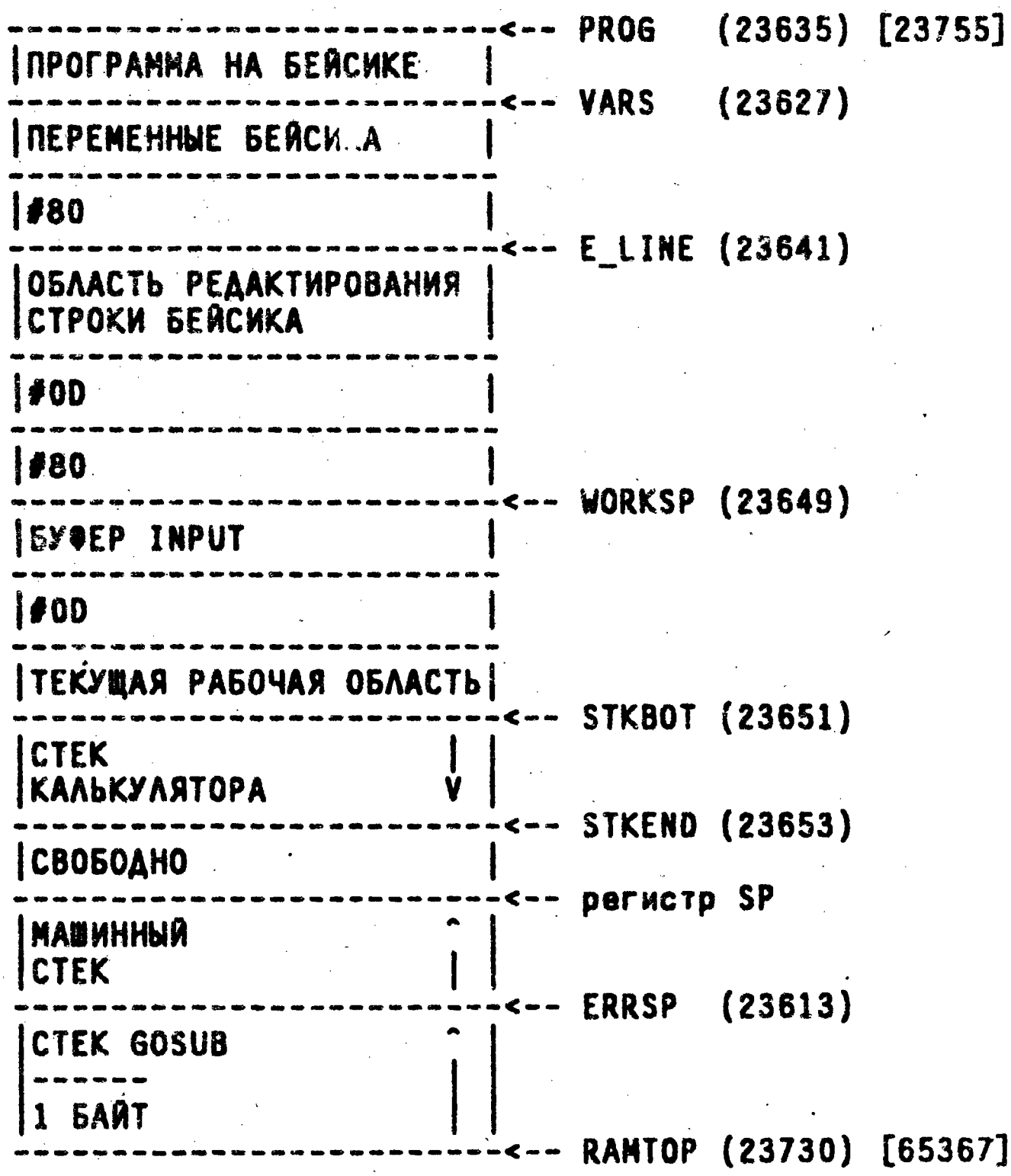

Рис.2. Область памяти системы БЕЙСИК

СИСТЕМНЫЕ ПЕРЕМЕННЫЕ - Эти ячейки памяти используются системой для запоминания необходимых для ее безошибочной работы данных. Таких как, например, адреса так называемых подвижных блоков памяти, информации о выполнении программы в БЕЙСИКе (т.е. какая строка выполняется, к какой должен осуществляться переход, появились ли какие-нибудь ошибки и т.п.). В этой области находятся также переменные, содержащие код последней нажатой клавиши, продолжительность "BEEP" кла-

виатуры и еще много других. Подробнее мы займемся этим позднее.

Непосредственно за системными переменными, которые кончаются адресом 23773, начинаются так называемые подвижные области памяти. Это означает, что адреса их начал и концов (а также длина) могут изменяться в зависимости от того, подключены ли какие-нибудь внешние устройства, какова длина программы на БЕЙСИКе, сколько образуется переменных и т.д. Адреса подвижных блоков находятся в системных переменных.

На рис.1 и рис.2 число за стрелкой означает адрес начала указываемого блока. Если адрес этот перемещается, то вместо адреса записано имя системной переменной, содержащей этот адрес, а в скобках - адрес этой переменной. В квадратных скобках находится значение этой переменной, которое устанавливается после включения компьютера (или выполнения RESET), но без подключения внешних устройств.

КАРТА МИКРОДРАЙВА - Если к Вашему компьютеру подключен ZX интерфейс-1, то с адреса 23734 до адреса, на 1 меньшего, чем содержимое CHANS, находится карта микродрайва - область, используемая как буфер для трансляции данных, как набор добавочных системных переменных и т.п. Если интерфейс не подключен, то эта область попросту не существует - переменная CHANS содержит адрес 23734. Она определяет начало блока памяти, в котором содержатся данные о существующих каналах.

ИНФОРМАЦИЯ О КАНАЛАХ - Необходима для безошибочного выполнения инструкций PRINT, LIST, INPUT и подобных им. В последней ячейке этой области находится #80 (128), определяющая конец этого блока (это - т.н. указатель конца).

ОБЛАСТЬ БЕЙСИКА - Область памяти содержит текст введенной программы. Адрес ее начала хранится в переменной PROG. Сразу за текстом программы (с адреса, указываемого переменной VARS) находится область, в которой интерпретатор размещает переменные, создаваемые программой. Она заканчивается указателем конца. Затем, начиная с адреса, содержащегося в переменной E_ LINE, находится область, используемая во время редактирования строки БЕЙСИКа, а также ввода директив с клавиатуры (т.е. когда в нижней части экрана мигает курсор и мывводим инструкцию на БЕЙСИКе). В конце этой области находятся 2 байта с содержимым 13 (ENTER) и 128 (конец этой области). Сразу после, начиная с адреса, указываемого переменной WORKSP, находится подобная область, но предназначенная для ввода данных во время выполнения интерпретатором инструкций INPUT (завершенной знаком ENTER). За буфером INPUT (который автоматически удаляется после выполнения этой инструкции) находится "текущее рабочее пространство" - место памяти, используемое для самых различных целей. Туда, между прочим, загружаются заголовки считанных с ленты программ. Туда считывается программа, размещаемая в памяти с помощью MERGE, прежде чем она будет присоединена к уже существующей программе. Эта область используется тогда, когда на определенное время требуется немного свободной памяти, но только во временное пользовани.. Однако, системе БЕЙСИКа принадлежит область памяти до ячейки, указываемой системной переменной RAMTOP. По этому адресу находится число #3 (62), которое устанавливает конец используемой БЕЙСИКом области. Двигаясь по памяти вниз, мы сталкиваемся с одним неиспользуемым байтом. Этот байт составляет совместно с байтом, указываемым с помощью RAMTOP, единое двухбайтное число (являясь его младшим байтом), необходимое для верной работы инструкции RETURN. Если во время ее выполнения стек GOSUB будет уже пуст, то это число

сыграет роль его продолжения. Но поскольку оно больше чем 15872 (62* 256), а строки БЕЙСИКа не имеют такой высокой нумерации, то это будет восприниматься как ошибка и сигнализироваться сообщением "RETURN без GOSUB". Сразу после этого байта (двигаясь все время вниз по памяти) начинается "стек GOSUB". В него заносятся номера программных строк, из которых были выполнены инструкции перехода к подпрограмме, чтобы интерпретатор знал точно, куда должна вернуть управление инструкция RETURN. Если интерпретатор не находится в подпрограмме (вызванной с помощью GOSUB), то этот стек просто не существует - в нем не записано ни одного значения. Ниже находится "машинный стек", непосредственно используемый микропроцессором. Оба этих стека откладываются в сторону уменьшения адресов памяти. Специальную роль играет системная переменная ERRSP. Процедура, обрабатывающая ошибку БЕЙСИКа (вызывается командой микропроцессора RST 8), помещает значение этой переменной в регистр SP, после чего выполняет RET, считывая таким путем последний записанный в стеке адрес (во время выполнения программы он равен обычно 4867). По этому адресу в ROM находится процедура, выводящая сообщение об ошибке.

ГРАФИКА, ОПРЕДЕЛЕННАЯ ПОЛЬЗОВАТЕЛЕМ - 168 байт, зарезервированных для определения знаков UDG (их можно ликвидировать, например, с помощью CLEAR 65535). Адрес последней ячейки памяти (равный 65535, если Ваш компьютер полностью исправен) запоминается в переменной P_RAMT. Если часть памяти RAM повреждена, то эта переменная содержит адрес последней исправной ячейки.

2. СПОСОБ ЗАПИСИ ПРОГРАММ НА ЛЕНТЕ.

Помните, что практически каждая программа имеет хотя бы процедуру -загрузчик, написанную на БЕЙСИКе, а если ее не имеет, то вообще не защищена.

Начнем со способа записи программ на ленте - припомним, что видно и слышно во время считывания какой-либо программы. В течении 1-2 секунд на экране видны широкие красно-синие полосы, а также слышен длительный звук. Это так называемый пилот, который позволяет компьютеру синхронизироваться с сигналом с ленты, который он будет считывать. Потом на момент появляются тонкие, мерцающие желто-фиолетовые полоски, свидетельствующие о том, что компьютер считывает в память информацию. Ее-17 байтов - это так называемый заголовок. Появляется надпись "BYTES", "PROGRAM:", "CHARACTER ARRAY:" или "NUMBER ARRAY:", а потом, после секундного перерыва, начинается второй пилот (более короткий), а после него считывается собственно программа. Займемся теперь заголовками, т.к. в них содержатся важнейшие данные о считываемых программах.

Заголовок, как мы уже вспомнили, содержит 17 байт. Пронумеруем их от 0 до 16 (см. рис.3). Нулевой байт обозначает тип блока. Он равен 0, если это - программа на БЕЙСИКе; 3 - если это блок машинного кода (записанный с помощью SAVE ".." CODE или SAVE ".." SCREEN$, который *обозначает точно то же, что и* SAVE ".." CODE 16384,6912). *Если же* этот заголовок предшествует набору, являющемуся массивом переменных БЕЙСИКа (записанный с помощью SAVE ".." DATA ..), то он равен 1 для числовых массивов или 2 для символьных.

Следующие 10 байт - это имя считываемого блока или текст, появляющийся после загрузки заголовка за надписью "PROGRAM:", "BYTES:" или другой.

Байты 11 и 12 содержат двухбайтовое число (первый байт младший), определяющее длину блока данных, к которому относится заголовок.

В зависимости от типа считываемого блока байты с 13 по 16 интерпретируются по-разному. Начнем с заголовков программ, написанных на БЕЙСИКе. Байты 13 и 14 содержат номер строки старта программы, если она была записана с помощью SAVE ".." LINE NR. Если программа была записана без опции LINE и после считывания не стартует автоматически, то значение этого числа больше 32767. Одним из способов нейтрализации защиты самостартующих программ на БЕЙСИКе является замена этих двух байтов в заголовке на число большее 32767.

Байты 15 и 16 содержат число, определяющее длину самой программы на БЕЙСИКе (т.к. SAVE ".." или SAVE ".." LINE записывает программу вместе со всеми переменными, т.е. содержимое памяти от байта, указываемого системной переменной PROG, до байта, определенного переменной E_LINE). Если от всей длины блока (байты 11 и 12) отнимем это число, то узнаем, сколько байтов в этом блоке занимают переменные БЕЙСИКа. Это все, если речь идет о заголовках программ на БЕЙСИКе.

В заголовках машинного кода ("BYTES") байты 15 и 16 не используются, зато 13 и 14 составляют двухбайтовое число, определяющее, по какому адресу надо загрузить следующий за заголовком блок.

В заголовках массивов из этих четырех байтов используется только 14-й байт, который представляет имя считываемого массива. Он записан так же, как и имена всех переменных БЕЙСИКа (в области от VARS до E_LINE), т.е. 3 самых старших бита обозначают тип переменной (здесь это числовой или символьный массив), а 5 младших битов - имя этой переменной.

0	ТИП БЛОКА:	0 - PROGRAM (ПРОГРАММА) 1 - NUMBER ARRAY (ЧИСЛОВОЙ МАССИВ) 2 - CHARACTER ARRAY (СИМВОЛЬНЫЙ МАССИВ) 3 - BYTES (ДАННЫЕ)
1		
2		
3		
4		
5		
6	ИМЯ	
7		
8		
9		
10		
11	ДЛИНА	СКОЛЬКО БАЙТ НЕОБХОДИМО СЧИТАТЬ С ЛЕНТЫ
12		
13	СТАРТ	BYTES: АДРЕС ЗАГРУЗКИ БЛОКА
14		PROGRAM: НОМЕР СТРОКИ СТАРТА ПРОГРАММЫ ARRAY: БАЙТ 14 - ИМЯ ПЕРЕМЕННОЙ 13 - НЕ ИМЕЕТ ЗНАЧЕНИЯ
15	ПРОГРАММА	PROGRAM: ДЛИНА ПРОГРАММЫ НА БЕЙСИКЕ (БЕЗ ПЕРЕМЕННЫХ)
16		BYTES, ARRAY: НЕ ИСПОЛЬЗУЕТСЯ

Рис.3. Заголовок блока

На листинге 1 представлена программа, реализующая процедуру, делающую возможным чтение заголовков программ, записанных на ленте. После точного ввода программы (внимательно проверьте количество пробелов в строках с инструкциями DATA) запустите ее с помощью RUN и подождите минуту. На экране появится информация об адресах начала и конца процедуры, а также о ее длине, которая должна составлять 284 байта. Если это не так, проверьте, все ли строки программы введены без ошибок. Если все данные были верны, то на Вашей ленте окажется процедура "ЧТЕЦ", благодаря которой Вы сможете прочесть каждый заголовок. После записи на ленту ее можно удалить из памяти.

Процедура "ЧТЕЦ" будет работать правильно, независимо от того, по какому адресу она будет загружена. Можете считать ее с помощью

LOAD "CZYTACZ" CODE AД EC

и затем запустить (даже многократно) с помощью

RANDOMIZE USR АДРЕС

После запуска процедура считывает по адресу 23296 (в буфер принтера) первый встреченный заголовок. Если из-за подключенных внешних устройств этот адрес Вас не устраивает, то его можно изменить. Для этого в строке с номером 200 надо заменить число #005B шестнадцатиричным адресом, по которому Вы хотели бы считывать заголовки (две первые цифры - младший байт). Чтобы после замены избежать проверки контрольной суммы в этой строке, в конце текста, взятого в кавычки, вместо пробела и 4-х цифр контрольной суммы поместите литеру "S" с 4-мя ведущими пробелами.

После считывания заголовка процедура читает заключенную в нем информацию и возвращается в БЕЙСИК, но считанного заголовка не уничтожает. Следовательно, если желаете просмотреть его дополнительно, можете сделать это с помощью функции PEEK.

Однако, чтения заголовка мало, чтобы смочь взломать блоки, записанные на ленте. Необходимо еще знать, что надо сделать с этими блоками, чтобы разместить их в памяти, не позволяя им при этом начать работу.

В случае блоков типа "BYTES" достаточно обычно загрузить их по адресу выше RAMTOP (т.е. выше ячейки памяти, указываемой системной переменной RAMTOP), например с помощью

CLEAR 29999: LOAD "" CODE 3000

Этот метод работает, если только считываемый блок не очень длинный (может иметь максимально до 40K). Более длинные блоки могут просто не поместиться в память - тогда необходимо разделить их на несколько более коротких частей. Скоро мы узнаем, как это делать.

Так же и в случае массивов. Их загрузка не вызывает затруднений - достаточно применить обычную в таких ситуациях инструкцию LOAD "" DATA.

Хуже, однако, выглядит считывание программ на БЕЙСИКе. Они обычно записываются с помощью SAVE ".." LINE ..., а в самом начале строки, с которой они должны выполняться, размещены инструкции, защищающие программы от останова. Простейшим решением является загрузка программы не с помощью LOAD "", а с помощью MERGE "", но этот способ не всегда дает результат. Из этой безнадежной ситуации существует два принципиальных выхода: подменить заголовок программы или использовать представленную ниже программу "LOAD/MERGE". Первый способ основан на замене записанного на ленте заголовка на такой же, но не вызывающий самозапуск программы. Можно с этой целью использовать программу

"COPYCOPY" - считать заголовок программы, нажать "BREAK", а затем инструкцией LET заменить ее параметр START на число большее 32767 (т.е. выполнить, например, LET I= 32768, если корректируемый заголовок был считан как первый набор). Модифицированный таким образом заголовок записываем где-нибудь на ленте. Убираем из памяти программу "COPYCOPY" и вводим LOAD "". Считываем только что сделанный заголовок, но сразу же после его окончания останавливаем ленту. Теперь в магнитофон вставляем кассету с программой так, чтобы считать только текст программы без ее заголовка.

Другой способ намного выгоднее. Вводим в память (с клавиатуры или с ленты) программу "LOAD/MERGE", приведенную на листинге 2. После запуска она начинает ждать первую программу на БЕЙСИКе, которая находится на ленте. Считывает ее совершенно так же, как инструкция LOAD "", но после загрузки не позволяет программе запуститься - выводит только сообщение "0 OK" с информацией, с какой строки считанная программа должна была стартовать.

На листинге 3 представлена процедура, которая образует эту программу из POKE-ов. Ее определяющей инструкцией является CALL 1821. Начиная с адреса 1541 в ROM находится процедура, интерпретирующая инструкции SAVE, LOAD, VERIFY, MERGE. CALL 1821 прыгает прямо в середину этой процедуры. Ее начальную часть (считывающую параметры этих инструкций) обходим, заменяя это строками 40 - 90 нашей процедуры: сначала с помощью RST 48 резервируем 34 байта в области WORKSP. Первые 17 байт заполняются процедурой из ROM после считывания параметров выполняемой инструкции БЕЙСИКа, а вторые - это место для считанного заголовка. Оба этих заголовка затем сравниваются перед тем как прочесть блок данных (проверяются соответствие имени и типа блока). Адрес свободного места переносим из регистра DE в IX; в строке 80 фиксируем системные переменные: T_ADDR, указываем, что нам нужна процедура LOAD (а не VERIFY, например); в первом байте заголовка для сравнения помещаем значение 255, означающее, что должен быть считан первый встреченный блок (то есть программа на БЕЙСИКе). Остальное берет на себя процедура из ROM, действующая идентично LOAD "". Разница основана на том, что после полного считывания программы вместо возврата по RET в интерпретатор БЕЙСИКа мы переносим значение переменной NEWPCC (номер строки, к которой должен быть переход) в PCC (номер последней выполняемой строки), а также не допускаем самозапуска программы - выполняем RST 8 с сообщением "0 OK". Благодаря этому переносу в выведенном сообщении будет информация о номере строки, с которой считанная программа должна была запуститься.

2.1. ЛИСТИНГИ ПРОГРАММ.

Листинг 1

```
 10 CLEAR 59999: LET POCZ=60000
 20 LET ADR=POCZ
 30 RESTORE: READ A,B,C,D,E,F
 40 DATA 10,11,12,13,14,15
 50 LET NR=200: RESTORE NR
 60 LET S=0: READ A$: IF A$="." THEN GO TO 130
 70 FOR N=1 TO LEN A$-5 STEP 2
 80 LET W=16*VAL A$(N)+VAL A$(N+1)
 90 POKE ADR,W: LET ADR=ADR+1: LET S=S+W
100 NEXT N
```

```
110 IF VAL A$(N TO)<>S THEN PRINT"ERR IN LINE";NR: STOP
120 LET NR=NR+10: GO TO 60
130 PRINT"OK"''"BEG:";POCZ'"END:";ADR-1'"LEN:";ADR-POCZ
140 SAVE "CZYTACZ" CODE POCZ,ADR-POCZ
150 REM
200 DATA "21920009E5DD21005BDDE51111 1246"
210 DATA "00AF37CD5605DDE130F23E02CD 1531"
220 DATA "011611C009DD7E00CD0A0CDDE5 1265"
230 DATA "D113010A00CD3C202A7E5CD1E5 1231"
240 DATA "D5ED537B5C010900CD3C20DD46 1346"
250 DATA "0CDD4E08CD2B2DCDE32DE1DD7E 1664"
260 DATA "00DD460FDD4E0DC5A7202BEB01 1292"
270 DATA "1900CD3C20D5DD4610DD4E0FCD 1361"
280 DATA "2B2DCDE32DD1C178E6C0206EC5 1848"
290 DATA "011300CD3C20C1CD2B2DCDE32D 1280"
300 DATA "185EFE03203601700009EB0111  836"
310 DATA "0018E60D449C75676F9D9B2073 1281"
320 DATA "616D65676F2070726F6772616D 1313"
330 DATA "75200D4175746F737461727420 1161"
340 DATA "2D206C696E696120D2EE1B0181 1239"
350 DATA "0009EB010900CD3C20C13E1FA0  997"
360 DATA "F660D7DD7E003D28033E24D73E 1383"
370 DATA "28D73E29D7E1227B5C3E0DD7C9 1538"
380 DATA "04081C2020201C000010181030  268"
390 DATA "100C000810384038047800D41   430"
400 DATA "647265732096C61646F77616E69 1357"
410 DATA "61200D5461626C69636120  862"
420 DATA "."
```

Листинг 2

```
1 REM LOAD/MERGE TS&RD 1987
2 FOR N=60000 TO 60025: READ A: POKE N,A: NEXT N
3 RANDOMIZE USR 60000
4 DATA 1,34,0,247,213,221,225,253,54,58,1,221,54,1,255,
       205,29,7,42,66,92,34,69,92,207,255
```

Листинг 3

```
 30    ORG   60000  ;LOAD/MERGE
 40    LD    BC,34
 50    RST   48
 60    PUSH  DE
 70    POP   IX
 80    LD    (IY+58),1
 90    LD    (IX+1),255
100    CALL  1821
110    LD    HL,(23618)
120    LD    (23621),HL
130    RST   8
140    DEFB  255
```

3. ЗАЩИТА ПРОГРАММ НА БЕЙСИКЕ.

Прочитанная программа как правило не должна выглядеть "нормальной". Например, в программе есть строка с номером 0 или строки, упо-

рядоченные по убыванию номеров; нельзя вызвать EDIT ни для какой строки; видно подозрительную инструкцию RANDOMIZE USR 0 или просто ничего не видно, т.к. программа не позволяет листать себя. Если в программе, куда Вы вламываетесь, видно что-то необычное, то лучше просматривать ее другим, несколько отличающимся от обычного способом: не с помощью LIST, а непосредственно используя инструкцию PEEK.

Однако, сначала мы должны узнать, каким образом размещен в памяти текст программы на БЕЙСИКе. Программа складывается из последовательных строк и так хранится в памяти. Отдельная строка выглядит таким образом:

```
 MSB LSB LSB MSB
-------------------------------
|2 байта|2 байта| . . . |#0D|
-------------------------------
 номер   длина     текст  ENTER
 строки  текста+1
```

Рис.4.

Она занимает не менее 5 (а точнее 6, так как текст пустым быть не может) байтов. Два первых означают ее номер, но он записан наоборот - в отличие от всех обычных двухбайтных чисел, хранящихся в памяти (MSB - старший байт, LSB - младший байт).

Следующие 2 байта - это длина строки, т.е. число символов, содержащихся в строке вместе с завершающим ее знаком ENTER (#0D). За этими байтами находится текст строки, заканчиваемый ENTER. Если мы введем, к примеру, такую строку:

```
10 REM BASIC
```

и запишем ее нажатием клавиши ENTER, то она будет записана в память как последовательность байтов:

```
-----------------------------------
|00|10|07|00|234|66|65|83|73|67|13|
-----------------------------------
  10     7   REM B  A  S  I  C ENTER
 номер длина        текст
```

Рис.5.

Параметр "длина строки" касается только ее текста. Следовательно, хотя строка занимает в памяти 11 байт, этот параметр указывает лишь на 7 байт: 6 байт текста и 1 байт - ENTER, заканчивающий строку.

Вам, наверное, уже понятно, на чем основан часто применяемый трюк со строкой, имеющей нулевой номер. Достаточно в первые два байта строки занести число 0 (с помощью POKE), чтобы эта строка стала нулевой строкой. Если мы хотим изменить номер первой строки в программе, то достаточно написать:

```
POKE 23755,X: POKE 23756,Y
```

и строка получит номер 256*X+Y. Независимо от его значения строка останется в памяти там, где была. Если, к примеру, введем

```
10 REM НОМЕР СТРОКИ 10
20 REM НОМЕР СТРОКИ 20

POKE 23755,0: POKE 23756,30
```

то первой строкой в программе будет номер 30, но она останется в памяти как первая, а на экране мы получим следующее:

```
30 REM НОМЕР СТРОКИ 10
20 REM НОМЕР СТРОКИ 20
```

Следовательно, чтобы начать разблокировать программу, в которой имеются нулевые строки или строки, упорядоченные по убыванию номеров, следует найти начала всех строк и в их поле "номер строки" последовательно размещать к примеру 10, 20, 30 ... В памяти строки располагаются одна за другой, следовательно, с обнаружением их начал Вы не будете иметь трудностей. Если X указывает адрес какой-нибудь программной строки, то следующий адрес равен

```
X+PEEK(X+2)+256*PEEK(X+3)+4
```

т.е. к адресу строки добавляется длина ее текста, увеличенная на 4 байта, т.к. именно столько занимают параметры "номер строки" и "длина строки".

Такой способ нахождения начала строки не действует к сожалению, когда применяется другой способ защиты - фальшивая длина строки. Он основан на том, что в поле "длина строки" вместо настоящего значения дается очень большое число - порядка 43-65 тысяч. Этот способ применяется очень часто, т.к. обычно делает невозможным считывание программы с помощью MERGE (т.е. так, чтобы не было самостарта). Делается это потому, что MERGE загружает программу с ленты в область WORKSP, а затем интерпретатор анализирует всю считанную программу строка за строкой: последовательно проверяет номер каждой из них, а затем размещает в соответствующем месте области, предназначенной для текста программы на БЕЙСИКе. Для каждой строки необходимо подготовить там соответствующее количество свободных байт, "раздвигая" уже существующий текст программы. Если в поле "длина строки" стоит очень большое число, то интерпретатор будет стараться освободить столько байтов в области текста программы, что это завершится сообщением "OUT OF MEMORY" или просто зависанием системы. Чтобы прочесть такую программу, не вызывая ее самозапуска, следует применить соответствующую отмычку. Например такую, как представленная в разделе 2 программа "LOAD /MERGE".

Дополнительным эффектом от применения фальшивой длины строки является невозможность ее корректировки путем занесения ее в поле редактора с помощью клавиши "EDIT". Ситуация выглядит аналогично: операционная система старается выделить место для этой строки в области редактирования строк БЕЙСИКа (от E_LINE до WORKSP), однако это требует слишком большого количества памяти и, следовательно, кончается только предупредительным звонком.

Если программа защищена этим способом, то адреса очередных строк приходится искать "вручную" или догадываться где они находятся, помня о том, что каждая строка кончается знаком ENTER (но не каждое число 13 означает ENTER).

Чтобы просмотреть программу на БЕЙСИКе, введите такую команду:

```
FOR N=23755 TO PEEK 23627+256*PEEK 23628:
PRINT N;"   ";PEEK N,CHR$ PEEK N AND PEEK N>31:
NEXT N
```

Она последовательно высветит адрес, содержимое байта с этим адресом, а также символ, имеющий этот код, если только это не управляющий символ (т.е. с кодом 0..31).

После смены нумерации строк и изменения их длин следующим способом защиты программ являются управляющие символы, не позволяющие пос-

ледовательно просматривать программу (хотя и не только это).

Вернемся к первому примеру (строка 10 REM BASIC). Текст строки складывается из 7 символов - ключевого слова REM (все ключевые слова - инструкции и функции, а также знаки <=, >= и <> имеют однобайтовые коды из диапазона 165..255. Если хотите их посмотреть - введите

```
FOR N=165 TO 255: PRINT N, CHR$ N: NEXT N
```

и ознакомьтесь), а также 5-ти литер и символа ENTER. Так бывает всегда, если в строке находится инструкция REM - все знаки, введенные с клавиатуры после этой инструкции, будут размещены в тексте строки без малейших изменений. Иначе выглядит ситуация, когда в строке находятся другие инструкции, требующие числовых параметров (а обычно так и бывает). Введем, например, строку

```
10 PLOT 10,9
```

и посмотрим, каким образом она запишется в память (лучше - вводя приведенную выше строку FOR N=23755 TO ...). Выглядит она так, как показано:

```
-----------------------------------------------------
|0|10|18|0|246|49|48|14|0|0|10|0|0|44|57|14|0|0|9|0|0|13|
-----------------------------------------------------
  10   18  PLOT 1  0     10        ,  9      9     ENTER
номер длина                  номер              номер
```

Рис.6.

Как видно, текст был модифицирован - после последней цифры каждого числа, выступающего в тексте строки параметр, интерпретатор сделал 6 байтов пространства и поместил там символ с кодом 14, а также 5 байтов, в которых записано значение этого числа, но способом, понятным интерпретатору. Это убыстряет (в определенной мере) выполнение программы на БЕЙСИКе, т.к. во время выполнения программы интерпретатор не должен каждый раз переводить числа из алфавитно-цифрового представления (последовательности цифр) в пятибайтное представление, пригодное для вычислений, а выбирает готовое значение из памяти после управляющего символа 14. Эта странная запись также дает большие возможности в деле затруднения доступа к программам. Во многих программах-загрузчиках присутствует такая строка:

```
0 RANDOMIZE USR 0: REM ...
```

На первый взгляд, после запуска эта программа должна затереть всю память. Однако, этого не происходит. После более внимательного просмотра (с помощью PEEK - строка FOR N=23755 ...) оказывается, что после USR 0 и символа CHR$ 14 совсем нет пяти нулей (а именно так выглядит число 0 в пятибайтовой записи. Если число - целое, то в пятибайтовой записи оно выглядит так:

```
байт 1      - 0
байт 2      - 0 для + или 255 для -
байт 3 и 4  - последовательно мл. и старш. байты
              (или дополнение до 2 для числа <0)
байт 5      - 0 ),
```

но стоит, к примеру, 0,0,216,92,0, что равнозначно числу 23770. Функция USR в этом случае осуществляет переход не по адресу 0, а именно по 23770. Это как раз адрес байта, находящегося в нашем примере сразу после инструкции REM. Обычно там находится программа-загрузчик, написанная на машинном языке.

Следующим управляющим символом, часто применяемым для защиты, яв-

ляется символ CHR$ 8 - "BACKSPACE" или "шаг назад". Высвечивание этого символа вызывает сдвиг позиции вывода на одну позицию влево. Следовательно, с помощью этого символа можно закрывать некоторые места в листинге, печатая там другой текст. Если, например, в памяти последовательно находятся знаки

```
LET A=USR 0: REM <<<<<<<<<<<<<<<<
LOAD ""
```

(< обозначает CHR$ 8), то инструкция LOAD "" и последующий текст закроют предшествующую инструкцию LET. Хотя на листинге теперь видна только инструкция LOAD, но дальнейшая часть программы загружается не ею, а машинной программой, вызываемой функцией USR. Такая защита применяется, например, в загрузчике программы BETA BASIC 1.0.

4. ЗАЩИТНЫЕ УПРАВЛЯЮЩИЕ СИМВОЛЫ.

В этом разделе мы познакомимся с остальными управляющими символами. Три из них касаются изменения места вывода, шесть - смены атрибутов.

4.1. CHR$ 6 -"COMMA CONTROL".

Этот символ (управляющая запятая) действует так же, как запятая, отделяющая тексты в инструкции PRINT, т.е. выводит столько пробелов (но всегда не менее одного), чтобы оказаться в колонке 0 или 16:

```
PRINT "1","2" и PRINT "1"+CHR$ 6+"2"
```

имеют идентичное значение.

4.2. CHR$ 22 -"AT CTRL".

Этот символ (AT управляющий) позволяет переносить позицию вывода в любое место экрана так же, как AT в инструкции PRINT. После этого символа должны следовать два байта, определяющие номер строки и номер колонки, в которой должен быть расположен следующий знак:

```
PRINT AT 10,7;"!"    равнозначно
PRINT CHR$ 22;CHR$ 10;CHR$ 7;"!"
```

Чтобы попробовать с помощью этих символов сделать листинг программы невидимым, введите:

```
10 RANDOMIZE USR 30000: REM НИЧЕГО НЕ ВИДНО!
```

После инструкции REM введите три пробела, а после "!" - две управляющие запятые. Их можно получить непосредственно с клавиатуры, нажимая последовательно клавиши EXTEND (или оба SHIFT вместе), чтобы получить курсор "E", а затем клавишу "6" (курсор сменит цвет на желтый) и DELETE - курсор перескочит к ближайшей половине экрана.

После ввода этой строки заменим три пробела после REM на знак AT 0, 0 с помощью

```
POKE 23774,22: POKE 23775,0: POKE 23776,0
```

и попробуем теперь посмотреть программу. На экране не появится текст всей строки - ее начальная часть закрыта надписью, находящейся после инструкции REM и знака AT CTRL. Такие же трудности возникнут, если эту строку перенести в зону редактирования (клавиша EDIT).

Координаты, заданные в символе AT CTRL, должны находиться в поле экрана, т.е. номер строки не должен быть более 21, а номер колонки - более 31. Указание больших значений в случае PRINT или LIST вызовет сообщение "OUT OF SCREEN" и отмену дальнейшего вывода. Если же листинг получен нажатием ENTER (автоматический листинг), то дальнейший вывод также будет прекращен, а кроме того в нижней части экрана появится мигающий знак вопроса - сигнал ошибки. Следовательно, это - на-

дежный способ защиты от просмотра любой программы.

4.3. CHR$ 23 - "TAB CTRL".

После этого символа (горизонтальная табуляция) следует перенос позиции вывода с помощью вывода пробелов - так же как и после управляющей запятой. Следовательно, он может использоваться аналогично для закрывания уже находящихся на экране текстов.

4.4. СИМВОЛЫ СМЕНЫ АТРИБУТОВ.

Эту группу управляющих символов составляют символы, меняющие атрибуты:

CHR$ 16 - INK CTRL
CHR$ 17 - PAPER CTRL
CHR$ 18 - FLASH CTRL
CHR$ 19 - BRIGHT CTRL
CHR$ 20 - INVERSE CTRL
CHR$ 21 - OVER CTRL

После каждого из этих символов обязателен 1 байт, уточняющий о каком атрибуте идет речь. После символов INK и PAPER могут быть числа 0.. 9; после FLASH и BRIGHT - 0, 1 и 8; после INVERSE и OVER - 0 и 1. Задание других значений вызывает сообщение "INVALID COLOR" и, естественно, прерывание просмотра программы.

Высвечивая программу, защищенную управляющими символами цветов, принимаем для себя следующую последовательность действий: если при просмотре текста программы мы встречаем код знака INK CTRL, то заносим в следующий за ним байт значение 7; после остальных управляющих цветами символов - 0. Кроме того, удаляем все мешающие знаки BACKSPACE (CHR$ 8), заменяя их на пробелы (CHR$ 32). Также ликвидируем знаки AT CTRL, заменяя с помощью POKE три байта символа на пробелы. После такой корректировки программу уже можно листать без всяких неожиданностей.

Если Вы хотите взломать программу-загрузчик, то его не обязательно очищать - важно узнать, что эта программа делает, каким образом загружает в память и запускает следующие блоки (а не стараться, чтобы она это делала "ладно" и была написана чисто и прозрачно). Это тем более важно, что пока не узнаешь точно программу, лучше не делать в ней никаких изменений - одна ловушка может проверяться другой. Поэтому наилучшим способом раскручивания программы является анализ ее работы шаг за шагом при просмотре последовательных байтов памяти:

```
0 BORDER 0: PAPER 0: INK 0: CLS: PRINT #0,
  "LOADING";: FOR N=0 TO 20 STEP 4: BEEP .2,N:
  NEXT N: LOAD "" CODE: PRINT AT 19,0;:
  LOAD "" CODE: PRINT AT 19,0;: LOAD "" CODE:
  PRINT AT 19,0;: LOAD "" CODE: PRINT AT 19,0;:
  LOAD "" CODE: RANDIMIZE USR 24064
```

Помните о правильной интерпретации очередных байтов памяти: сначала два байта номера строки, потом два байта ее длины (которая может быть фальшивой), затем текст с инструкцией БЕЙСИКа и ее параметрами. За каждым числом дописываются CHR$ 14 и пять байтов, содержащих значение этого числа. За параметрами - двоеточие и следующая инструкция (или ENTER) и новая строка программы.

Это было бы все, если речь шла только об управляющих символах. Но

есть еще одна вещь, которую требуется объяснить, чтобы Вы не имели проблем с чтением БЕЙСИКа. Речь идет об инструкции DEF FN. Введите, а затем внимательно просмотрите такую строку:

```
10 DEF FN A(A,B$,C) = A+C
```

Кажется, что она должна занять в памяти 19 байтов (номер строки, ее длина, ENTER и 14 введенных символов), но это не так. Интерпретатор после каждого параметра функции поместил знак CHR$ 14 и зарезервировал зачем-то следом еще 5 байтов. Введите

```
PRINT FN A(1,"125",2)
```

и снова внимательно просмотрите содержимое памяти с адреса 23775. После первого параметра в определении функции далее находится CHR$ 14, но после него последовательно расположились 0,0,1,0,0, что в пятибайтовой записи означает число 1. После третьего параметра также находится CHR$ 14 и байты, содержащие число 2. После параметра B$ также находится значение используемого параметра: CHR$ 14 и пять байтов. Первый из них не имеет значения, второй и третий содержат адрес, по которому находилась цепочка символов "125" (вызов функции был осуществлен в директивном режиме, следовательно, этот адрес относится к области редактирования строки БЕЙСИКа), а байты 4 и 5 - это длина цепочки. В нашем случае она равна трем.

Помните об этом, читая БЕЙСИК с помощью PEEK, а не LIST. Иногда случается, что именно в этих байтах, зарезервированных для действительных аргументов функции, скрыты проверки, определяющие работоспособность программы или даже - машинная программа, загружающая последующие блоки. Пример - BETA BASIC 1.0.

Теперь - немного о программах-загрузчиках. Их задачей является считывание и запись всех блоков, составляющих программу. Обычно они делают это способом, максимально затрудняющим понимание их работы: так, чтобы запуск программы другим способом, а не через загрузчик (или на практике - взлом программы) был невозможен. Посмотрим на загрузчики фирмы ULTIMATE (например: ATIC, ATAC, KUIGHTLORE, PENTAGRAM, NIGHT SHADE и т. д.). Выглядят они примерно так:

```
FOR N=23755 TO PEEK 23627+256*PEEK 23628: PRINT N;
"   "; PEEK N, CHR$ PEEK N AND PEEK N>31: NEXT N
```

После такого загрузчика на ленте находится 5 следующих блоков: экран, закодированный главный блок программы и три коротеньких блока, защищающих программу. Первый - однобайтовый (код инструкции JP HL), второй - из нескольких байтов (это - процедура, декодирующая всю программу) и третий - двухбайтовый, загружаемый по адресу 23672 или в переменную FRAMES. Значение этой переменной увеличивается на 1 через каждые 1/50 секунды. В машинной программе, запущенной с помощью RANDOMIZE USR 24064, ее значение проверяется и, если оно отличается от того, каким должно быть (что означает остановку программы после загрузки на какое-то время), выполняется обнуление памяти компьютера. Взлом программ такого типа весьма прост. Достаточно загрузить все блоки за исключением последнего, а после просмотра программы или же проведения в ней необходимых изменений (например - посредством POKE) ввести

```
LOAD "" CODE: RANDOMIZE USR 24064
```

(но обязательно - в одной строке) чтобы запустить игру. Отдельной работой является декодирование программы или запуск процедуры, декодирующей так, чтобы она по завершению работы вернулась в БЕЙСИК. У читателей, знающих АССЕМБЛЕР, это не должно вызвать затруднений, однако

с учетом распространенности этого типа защиты, особенно в пользовательских программах (например - ART STUDIO или THE LAST WORD) мы еще вернемся к этой теме.

5. ЗАЩИТА ЗАГРУЗЧИКОВ.

Все игры имеют очень хорошо защищенную программу, написанную на БЕЙСИКе, т.к. это - важнейший (с точки зрения действенности защиты) элемент всей программы - ведь с БЕЙСИКа начинается считывание всей программы. Если БЕЙСИКовский загрузчик защищен слабо, то взлом всей программы значительно облегчается. Примером этого являются загрузчики фирмы ULTIMATE, представленные в предыдущем разделе. Одним из способов снятия защиты загрузчиков является их считывание с помощью программы "LOAD/MERGE" (см. раздел 2). Однако иногда лучше поместить этот загрузчик не в памяти, предназначенной для БЕЙСИКа, а выше RAMTOP, чтобы можно было спокойно просматривать его, не опасаясь возможности случайных изменений в нем.

Для этого имеется очень действенный метод: считывание программы на БЕЙСИКе как блока машинного кода по удобному для нас адресу. Чтобы добиться этого, требуется знать длину программы, которую мы хотим считать (можете использовать процедуру "CZYTACZ" из раздела 2), хотя можно обойтись и без длины. Кроме того требуется немного свободного места на магнитной ленте. Этот способ основывается на обмане инструкции LOAD путем подмены заголовка.

На свободном месте ленты записываем заголовок блока кода с помощью SAVE "BAS" CODE 30000,750, если знаем, что длина программы составляет 750 байт. Если не знаем, то подаем соответсвенно большее значение - порядка нескольких килобайт или даже десятков килобайт, хотя программа может иметь всего лишь 100 байтов длины. На ленте записываем только сам заголовок, прерывая запись после этого нажатием клавиши "BREAK". Теперь устанавливаем ленту точно перед записанным заголовком, а ленту с программой - сразу после заголовка программы, но перед требуемым блоком данных. Вводим

```
CLEAR 29999: LOAD "" CODE
```

и считываем заголовок. Сразу после его считывания нажимаем STOP на магнитофоне, заменяем кассету и вновь нажимаем PLAY (все это время компьютер ждал блок данных). Теперь считывается программа на БЕЙСИКе, но по адресу 30000 (выше RAMTOP). Если мы указали в заголовке завышенную длину программы, то считывание кончается "TAPE LOADIND ERROR", но это не мешает - теперь уже можно смотреть считанную программу любым способом, даже набирая с этой целью программу на БЕЙСИКе.

Кроме этого метода существует и второй, но для того чтобы им воспользоваться, обязательно знание АССЕМБЛЕРа (а пользоваться стоит, т. к. он дает большие возможности в раскрытии программ).

Очень часто (особенно в новейших программах) встречаются блоки, записанные и считываемые в память компьютера без заголовка. Это достаточно оригинальная и эффективная мера защиты обычно отпугивает начинающих, но раскрытие такой программы вовсе не является трудным. Вся тайна основана на использовании хранящихся в ROM SPECTRUM процедур, используемых инструкциями LOAD, SAVE, VERIFY и MERGE.

По адресу #0556 (1366) находится процедура LOAD_BYTES, считывающая с магнитофона блок данных или пилот и следующую за ни минформацию. *При этом неважно - будет ли это заголовок или требуемый блок данных, которые следует разместить где-то в памяти.*

Начнем все же с начала. Каждая защищенная программа начинается с загрузчика, написанного на БЕЙСИКе. Программа, применяющая загрузку без заголовков (с помощью процедуры 1366 или другой), должна быть написана в машинном коде, как всякая процедура, обслуживающая магнитофон. Чаще всего эта программа помещается в одной из строк БЕЙСИКа, например после инструкции REM или в области переменных БЕЙСИКа. После считывания загрузчик на БЕЙСИКе запускается и выполняет инструкцию RANDOMIZE USR ..., инициируя тем самым работу машинной программы.

Процедура LOAD_BYTES требует соответствующих входных параметров. Они передаются в соответствующих регистрах микропроцессора. Так, в регистре IX указываем адрес, по которому хотим прочесть блок данных, а в регистре DE - длину этого блока. В аккумулятор помещаем 0, если хотим считать заголовок и 255, если это - блок данных. Кроме того, указатель переноса (CARRY) устанавливаем в 1, т.к. иначе процедура 1366 вместо LOAD выполнит VERIFY. Ниже дан пример процедуры, загружающей с ленты образ экрана без заголовка:

```
        LD   IX,16384 ;адрес считывания
        LD   DE,6912  ;длина блока
        LD   A,255    ;блок, а не заголовок
        SCF           ;LOAD, а не VERIFY
        CALL 1366     ;вызов LOAD_BYTES
        RET           ;возврат из п/п
```

Процедура 1366 в случае ошибки считывания не выводит сообщение "TAPE LOADING ERROR", но существует еще одна процедура загрузки, которая это делает. Она находится по адресу 2050 и выглядит так:

```
2050 CALL 1366 ;считывание блока данных
2053 RET  C    ;возврат, если не было ошибки
2054 RST  8    ;иначе RST 8 с сообщением
2055 DEFB 26   ;"TAPE LOADING ERROR"
```

После возврата из процедуры 1366 указатель переноса содержит информацию о правильности считывания блока. Если он равен 0, то это сигнализирует об ошибке. Некоторые загрузчики используют именно процедуру 2050, а не 1366.

Иногда загрузчики не пользуются ни той, ни другой процедурой, а заменяют их собственной. Однако, она часто является просто переделкой 1366: например, блоки загружаются от больших адресов к меньшим или загрузка идет с другой скоростью. Такую программу следует анализировать с помощью дизассемблера (например MONS), сравнивая ее фрагменты с тем, что находится в ROM.

Сейчас мы объясним, как использовать процедуры из ROM для считывания из БЕЙСИКа под любой адрес, а не в область, предназначенную для него: сначала с помощью "CZYTACZ" прочитаем заголовок программы, которую мы хотим взломать, и запомним ее длину (т.е. длину всего блока - программы вместе с переменными). Затем введем соответствующую программу на ассемблере, которая прочитает БЕЙСИК-программу по адресу, который мы зададим (выше RAMTOP):

```
        LD  IX,addr
        LD  DE,len
        LD  A,255
        SCF
        JP  2050
```

Так же, как и при подмене заголовка, можем указать завышенное значение длины программы (если не знаем истинного значения), но в

этом случае чтение завершится сообщением "TAPE LOADING ERROR". Чтобы не считывать АССЕМБЛЕР всякий раз для ввода вышеприведенной программы, создадим эту программу из БЕЙСИКа с помощью POKE:

```
 10 INPUT "АДРЕС ЧТЕНИЯ ?"; A
 20 RANDOMIZE A: CLEAR A-1
 30 LET A=PEEK 23670: LET B=PEEK 23671
 40 LET ADR=256*B+A
 50 INPUT "ДЛИНА ?"; C
 60 RANDOMIZE C: LET C=PEEK 23670
 70 LET D=PEEK 23671
 80 FOR N=ADR TO ADR+11
 90 READ X: POKE N,X: NEXT N
110 DATA 221,33,A,B,17,C,D,62,255,125,2,8
120 RANDOMIZE USR ADR
```

Устанавливаем ленту с обрабатываемой программой за ее заголовком, затем запускаем вышеприведенную программу, вводим данные и включаем магнитофон. Результат аналогичен тому, который получаем при подмене заголовков, но первым же видимым достоинством этого способа является то, что мы не создаем беспорядка на кассетах.

В завершение стоит вспомнить еще об одной процедуре, размещенной в ROM по адресу 1218. Это процедура SAVE_BYTES, обратная LOAD_BYTES, т.е. записывающая на ленту блок с заданными параметрами: перед ее вызовом в регистре IX размещаем адрес, с которого начинается записываемый блок, DE должен содержать длину этого блока, в аккумуляторе помечаем, должен ли это быть заголовок (0) или же собственно блок (255). Состояние указателя CARRY значения не имеет.

6. ИСПОЛЬЗОВАНИЕ СИСТЕМНЫХ ПРОЦЕДУР.

В предыдущем разделе были представлены процедуры из ROM: LOAD_BYTES и SAVE_BYTES. Здесь рассказывается, как эти процедуры используются для защиты программ. Займемся блоками машинного кода, которые запускаются удивительным образом.

Первым таким способом является прикрытие загрузчика блоком, который им загружается. Такая защита, к примеру, применяется в игре "Три недели в Парадиз-сити". Проследим способ чтения этой программы, не вызывающий ее автоматического запуска.

Начнем, как обычно, с БЕЙСИКа. Оказывается, практически единственно важной инструкцией является RANDOMIZE USR. Лучше всего восстановить эту процедуру, начиная с адреса запуска (PEEK 23627+256*PEEK 23628), т.е. в нашем случае с адреса 24130. Подобные процедуры используют известную нам процедуру 1366, считывающую блоки без заголовка. Так и в этом случае, но перед ее вызовом процедура загрузки переносит сама себя в конец памяти (по адресу 63116)с помощью команды LDIR и переходит туда сама по команде JP:

```
24130 DI                ;запрет прерываний
24131 LD   SP,0         ;LD SP,65536
24134 LD   HL,(23627)   ;в HL адрес на 28 больше, чем
24137 LD   DE,28        ;значение переменной VARS
24140 ADD  HL,DE        ;
24141 LD   DE,63116     ;в DE - адрес, а в
24144 LD   BC,196       ;BC - длина блока
24147 LDIR              ;
24149 JP   63116        ;продолжение выполнения
```

Теперь считывается картинка, а затем - главный блок данных:

```
63116 LD   IX,16384 ;подготовка загрузки картинки
63120 LD   DE,6912  ;на экран с помощью процедуры
63123 LD   A,255    ;LOAD_BYTES из ROM
63125 SCF           ;
63126 CALL 1366     ;
63129 LD   IX,26490 ;параметры главного блока
63133 LD   DE,38582 ;программы, который при своей
63136 LD   A,255    ;загрузке затирает эту процедуру
63138 SCF           ;
63139 CALL 1366     ;
--------------------
63142 JP   NZ,+79   ;после возврата из процедуры
63144 CP   A        ;1366 здесь уже находится
63146 CALL 65191    ;другая программа
```

Как Вы знаете, каждая инструкция CALL заносит в машинный стек адрес, с которого продолжает работать программа после выхода из подпрограммы. В этом загрузчике после выполнения второй инструкции CALL 1366 в стек заносится адрес команды, следующей за CALL, т.е. 63142, и процедура загрузки самым обычным способом затирает сама себя, т.к. считывает байты с магнитофона в ту область памяти, где она была размещена. Существенное значение имеет способ запуска считываемой программы: процедура 1366 кончается, естественно, инструкцией RET, которая означает переход по адресу, записанному в стек. В нашем случае это 63142. В процессе считывания программы процедура, которая находилась там, была заменена считанной программой. Но микропроцессор этого не замечает - он возвращается по адресу, с которого выполнен CALL 1366, не обращая внимания, что там находится уже совершенно другая программа. Это схематично представлено на рис.7:

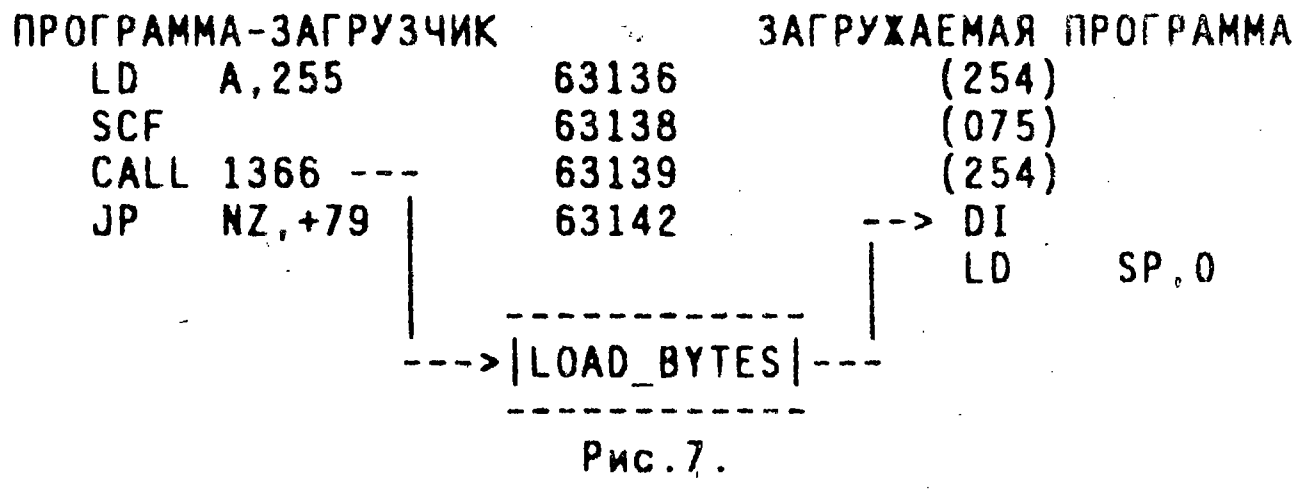

Рис.7.

С левой стороны расписано содержимое памяти до, а с правой - после считывания программы.

Возникает вопрос, как распознать защиту данного типа и как ее ликвидировать. Начинаем, естественно, с БЕЙСИКа. Считываем загрузчик (в АССЕМБЛЕРе) и определяем адреса окончаний считанных блоков, добавляя к адресу начала (регистр IX) длину блока (регистр DE). Если какой-либо из блоков накрывает процедуру загрузки, то это означает, что программа считывается и запускается именно этим способом.

Дальше все просто. Достаточно, опираясь на данные о блоках (адрес и длину), написать коротенькую процедуру, загружающую интересующий нас блок кода, или подготовить соответствующий заголовок, затем с по-

мощью CLEAR ADR установить соответствующим образом машинный стек (чтобы считываемая программа не уничтожила стек) и, наконец, считать программу. После выполнения в ней необходимых изменений записываем ее на ленту, но таким же образом, каким был записан оригинал (длина должна совпадать прежде всего). Если этот блок был без заголовка (а так и есть в нашем случае), то записываем его обычным SAVE "..." CODE ..., но опуская заголовок, т.е. включаем магнитофон в перерыве между заголовком и блоком кода. Также можно попробовать запустить считанный блок переходом на требуемый адрес командой RANDOMIZE USR ..., но это не всегда может получиться. В игре "Три недели в Парадиз-сити" этим адресом, конечно, будет 63142 и, как Вы можете убедиться, этот метод срабатывает.

Другим интересным способом запуска блоков машинного кода является считывание программы в область машинного стека. Этим способом можно запускать блоки машинного кода, загружая их просто через LOAD "" CODE. Этот метод схематично представлен на рис.8.

Указатель стека (регистр SP) принимает показанное на рис.8 состояние в процессе выполнения процедуры 1366 (вызванной из БЕЙСИКа через LOAD "" CODE). Способ запуска программы весьма прост. Адрес считывания блока рассчитан так, что блок считывается на машинный стек именно с того места, в котором находится (записанный интерпретатором БЕЙСИКа) адрес возврата из инструкции LOAD "" CODE (он тогда равен значению системной переменной ERRSP-2) или прямо из процедуры LOAD_BYTES (в этом случае он равен ERRSP-6). Тогда два первых байта программы обозначают адрес ее запуска. Этот способ очень похож на предыдущий, за исключением того, что там подменялась процедура загрузки, а здесь - адрес возврата из этой процедуры или просто адрес возврата из инструкции LOAD. После считывания блока кода микропроцессор считывает содержимое стека и переходит по прочитанному адресу, который только что появился в памяти вместе со считанной программой. По этому адресу в программе находится начало процедуры загрузки ее последующих блоков - это видно на рис.8.

```
рег.SP->65358     1343
         65359
         65360    2053
         65361          <--
         65362    7030      |  65368     |
         65363              |            |
         65364    4867      |     ?      |
         65365              |            |
         65366      ?       |     ?      |
RAMTOP->65367     62         |    62      |
         65368               | LD IX,     |
         65369               |       ...  |
         65370               | LD DE,     |
                             |       ...  |
```

Рис.8.

Метод обхода этой защиты также весьма прост. Достаточно заменить RAMTOP на более низкое значение, а затем прочитать блок кода, который благодаря этому не запустится. Ситуация может осложниться, если блок очень длинный (что редко, но случается) - тогда мы должны поступить с

ним так же, как с любым длинным блоком, но помнить, с какого адреса он запускается.

Займемся теперь расчленением блоков кода с длиной, превышающей 42К. Взлом блоков этого типа основан на разделении их на такие фрагменты, чтобы в памяти еще осталось место MONSa или другого дизАССЕМБЛЕРа; исправлении этих фрагментов и их склеивании в одно целое (или написание новой программы загрузки). Обычно достаточно разделить длинный блок на две части. Чтобы получить первую, используем процедуру 1366, но с другими параметрами (не с теми, которые требует разделяемый на части блок). Ранее из процедуры загрузки или (если такой нет) из заголовка этого блока получаем его длину и адрес загрузки. В качестве адреса загрузки указываем величину выше RAMTOP, а длину - 16 К. Считываем теперь этот блок через CALL 1366 или CALL 2050, но в обоих случаях появившееся сообщение "TAPE LOADING ERROR" не даст нам никакой информации о верности считывания - загружаем лишь часть блока (следовательно - без контрольного байта). Считанную таким образом первую часть блока записываем на ленту и сразу же приступаем ко второй части. Ее считывание труднее, но тоже возможно, несмотря на ограничение по памяти. Достаточно использовать тот факт, что у SPECTRUM существует 16К ROM, запись в которую просто невозможна. Итак, вызываем процедуру 1366 с адресом чтения 0, и начальные 16К считываемого блока будут потеряны, а в память RAM считаются только следующие 32К или меньше (в зависимости от длины блока). Считываемый блок займет в памяти адреса с 16384 и дальше, заходя на системные переменные и оставляя без изменения лишь те байты, адреса которых больше длины блока. Поэтому необходимо позаботиться о том, чтобы машинный стек, а также написанную нами процедуру загрузки разместить в конце памяти. Надо также помнить о том, что система БЕЙСИКа будет уничтожена, и записать считанный блок на ленту можно будет только процедурой, написанной на АССЕМБЛЕРе. Кроме того, в промежутках времени между считыванием фрагмента блока и его записью нельзя разблокировать прерывания, т.к. они изменяют содержимое ячеек памяти с адресами 23552..23560, а также 23672..23673. Чтобы выполнить последнее условие, войдем в середину процедуры 1366, благодаря чему после считывания блока не будет выполнена процедура 1343. Именно она и разблокирует прерывания. С помощью CLEAR 64999 переносим машинный стек, а с адреса 65000 помещаем процедуру загрузки:

```
        ORG  65000
        LD   IX,0          ;адрес считывания
        LD   DE,DL         ;длина блока
        LD   A,255         ;подготовка к
        SCF                ;считыванию блока
        INC  D             ;таким способом
        EX   AF,AF'        ;заменяем начало
        DEC  D             ;процедуры 1366,
        DI                 ;а затем входим
        LD   A,15          ;в ее середину
        OUT  (254),A       ;
        CALL 1378          ;
        LD   A,0           ;черная рамка
        JR   C,OK          ;будет означать
        LD   A,7           ;верное считывание,
OK      OUT  (254),A       ;белое - ошибочное
```

```
CZ  LD   A,191         ;охидаем нахатия
    IN   A,(254)       ;"ENTER"
    RRA                ;
    JR   C,CZ          ;
    LD   IX,0          ;запись считанного
    LD   DE,DL-16384   ;блока кода
    LD   A,255         ;на ленту,
    CALL 1218          ;а такхе
    LD   HL,64999      ;инициализация
    JP   4633          ;системы БЕЙСИКа
```

Вместо того, чтобы считывать АССЕМБЛЕР и вводить эту программу, мохно запустить программу на БЕЙСИКе, представленную на листинге:

```
10 CLEAR 64999
20 INPUT "ДЛИНА БЛОКА?"; DL: RANDOMIZE DL:
   LET X=PEEK 23670: LET Y=PEEK 23671: LET S=0
30 FOR N=65000 TO 65053: READ A: LET S=S+A:POKE N,A: NEXT N
40 IF S<>5254+2*(X+Y)-64 THEN PRINT "ОШИБКА В ДАННЫХ": STOP
50 PRINT "ВСЕ ДАННЫЕ О'К, ВКЛЮЧИТЕ МАГНИТОФОН"
60 RANDOMIZE USR 65550
70 DATA 221,33,0,0,17,X,Y,62,2,55,55,
        20,8,21,243,62,15,211,254
80 DATA 205,98,5,62,0,56,2,62,7,211,254,
        62,191,219,254,31,56
90 DATA 249,221,33,0,0,17,X,Y-64,62,255,205,
        194,4,33,231,253,202,25,18
```

Как выглядит разделение блока? Устанавливаем ленту на блоке кода, который хелаем разделить. Если он имеет заголовок, то его опускаем. Запускаем процедуру и включаем магнитофон. Не пугайтесь, если в определенный момент увидите, что программа считывается на экран - так и долхно быть. После загрузки блока цвет рамки сигнализирует правильность считывания: если рамка черная, то все в порядке, если белая - была ошибка. Влохите в магнитофон другую кассету, включите запись и нахмите "ENTER". Программа запишется на ленту, а потом процедура вернет управление в БЕЙСИК, инициализируя систему с сообщением "(C) 1982 ...", но не очищая память (уничтохается только область от начала экрана до адреса примерно 24000). Теперь, подготавливая заголовок или записывая коротенькую процедуру, мохно прочесть полученный блок по любому адресу.

Когда Вы найдете то, что искали и захотите запустить измененную программу, то придется немного помучиться и склеить разделенную программу или написать для нее новую процедуру загрузки. Если программа заполняла полностью 48K памяти RAM, то возможен только второй метод.

Если надо соединить блоки, то достаточно написать процедуру, похохую на разделяющую. Она долхна считывать первый блок по адресу 16384, второй - сразу за ним, а затем записывать вместе как один блок.

В этом разделе был описан способ запуска блоков машинного кода путем считывания в область машинного стека. Чтобы такой блок запустился, достаточно ввести инструкцию LOAD "" CODE, которая запустит его. Чтобы убедиться в этом на практике, введите и запустите программу с листинга:

```
10 CLEAR 65361: LET S=0
20 FOR N=65362 TO 65395: READ A: POKE N,A:LET S=S+A: NEXT N
30 IF S<>3437 THEN PRINT "ОШИБКА В СТРОКЕ DATA": STOP
40 SAVE "BEEP" CODE 65362,34
50 DATA 88,255,3,19,0,62,221,229,6,8,
        197,17,30,0,33,112,5,205
60 DATA 181,3,33,0,8,43,124,181,32,251,
        193,16,235,221,225,201
```

Она запишет на ленте короткий блок машинного кода, который будет запускаться самостоятельно. После записи этого блока освободите память с помощью RANDOMIZE USR 0 или RESET (по возможности переставьте RAMTOP на нормальное значение с помощью CLEAR 65367) и прочтите его, введя LOAD "" CODE. Цель этого блока - проинформировать о том, что он запустился: он это делает несколькими звуковыми сигналами. Вы услышите их сразу после считывания как только с рамки исчезнут гранатово-желтые полосы.

7. ДЕКОДИРОВАНИЕ ЗАКОДИРОВАННЫХ БЛОКОВ ТИПА "BYTES".

Что означает, что программа или блок закодированы? Кодирование - это род весьма простого шифра, делающего невозможным правильную работу программы. Для ее запуска служит специальная декодирующая процедура, которая находится в этой же программе, а также (что самое важное) не закодирована. Так для чего же служит кодирование? Это просто очередное затруднение доступа к тексту программы после того, как все предшествующие предохранители взломаны и программа считана без запуска. В этом случае в памяти лежит "полуфабрикат", который только после переработки декодирующей процедурой становится программой. Кодирование может быть основано, например, на инверсии всех байтов в программе. Хорошим примером является программа "ART STUDIO". Ее БЕЙСИКовская часть практически никак не защищена, но главный блок программы ("STUDIO-MC" CODE 26000,30672) частично закодирована. Что сделать, чтобы раскодировать его? Просто найти адрес, с которого этот блок запускается. В случае "ART STUDIO" этот адрес - 26000. Там находится инструкция JP 26024, которая осуществляет переход к декодирующей процедуре. Вот ее текст:

```
26024 21C165    LD   HL,26049; Запись адреса
26027 E5        PUSH HL      ; 26049 в стек.
26028 21C165    LD   HL,26049; Адрес начала
26031 11476C    LD   DE,27719; Адрес конца
26034 7E        LD   A,(HL)  ; Выбор байта из
26035 D622      SUB  34      ; памяти, перекоди-
26037 07        RLCA         ; рование его с по-
26038 EECC      XOR  #CC     ; мощью SUB, RLCA,
26040 77        LD   (HL),A  ; XOR и запись.
26041 23        INC  HL      ; Следующий адрес
26042 B7        OR   A       ; Проверка признака
26043 ED52      SBC  HL,DE   ; конца, возврат в
26045 19        ADD  HL,DE   ; цикл или выход по
26046 20F2      JP   NZ,26034; адресу, записан-
26048 C9        RET          ; ному в стеке или
26049 B2EDF1                 ; по 26049
```

Сначала процедура помещает в стек адрес 26049, после чего в регистр HL заново загружается 26049 (начало закодированного блока), а в

DE -27719 (адрес последнего закодированного байта). Затем в цикле последовательно декодируются байты - инструкции SUB 34, RLCA и XOR #CC являются ключем, с помощью которого расшифровывается эта часть программы. Наконец, проверяется условие достижения адреса 27719, как последнего декодируемого. Выполняется инструкция RET, но последним записанным в стек адресом является не адрес возврата в БЕЙСИК, а записанный в начале процедуры с помощью PUSH HL адрес 26049 (или адрес только что раскодированного блока). Следовательно, происходит его запуск.

Обратим внимание, какие инструкции осуществляют дешифрацию: никакая из них не теряет ни одного бита. Вычитание осуществляется по модулю 256, и для для двух разных входных данных результаты тоже различны. RLCA заменяет значения битов 7, 6, 3 и 2 на противоположные. Другими инструкциями, имеющими те же самые свойства, являются, например, ADD, INC, DEC, RRCA, NEG, CPL, но не OR или AND. Как раскодировать этот блок? Проще всего ввести в БЕЙСИКе

POKE 26027,0 (код инструкции NOP)

ликвидируя тем самым инструкцию PUSH HL (RET в конце программы приведет в БЕЙСИК), и выполнить RANDOMIZE USR 26000. Стоит, однако, помнить о том, что декодирующая программа может проверяться другим фрагментом программы. В "ART STUDIO" так и есть. Вот дальнейшая часть программы:

```
26289  67      LD    H,A        ;в A уже находится
26290  6F      LD    L,A        ;значение 0
26291  E5      PUSH  HL         ;запись его в стек
26292  3AA865  LD    A,(26024)  ;и проверка содер-
26295  FE21    CP    #21        ;жимого ячеек
26297  C0      RET   NZ         ;с адресами
26298  2AA965  LD    HL,(26025);26024..26027
26301  B7      OR    A          ;и если оно другое,
26302  11C165  LD    DE,26049   ;то стирание памяти
26305  ED52    SBC   HL,DE      ;с помощью RET NZ
26307  C0      RET   NZ         ;(под адрес 0)
26308  3AAB65  LD    A,(26027)  ;если все нормально
26311  FEE5    CP    #E5        ;то снятие адреса 0
26313  C0      RET   NZ         ;и нормальный
26314  E1      POP   HL         ;выход
26315  C9      RET              ;
```

Этот фрагмент программы проверяет, наверняка ли декодирующая процедура запустила всю программу, и если нет, то с помощью RET NZ стирает память (т.к. в стеке записан 0). Что делать в этом случае? Можно ликвидировать и эти меры защиты, но в некоторых программах эта мера не помохет. Тогда остается другой выход: снова закодировать программу, т.е. сделать обратное действие. В нашем случае следовало бы выполнить

```
XOR   #CC
RRCA
ADD   34
```

В "ART STUDIO" неизвестно для чего была помещена кодирующая программа. Она находится по адресу 26003 и выглядит следующим образом:

```
26003  21C165  LD    HL,26049 ;адрес начала
26006  11476C  LD    DE,27719 ;адрес конца
26009  7E      LD    A,(HL)   ;выборка байта из
```

```
26010  EECC     XOR   #CC        ;памяти, кодирование
26012  0F       RRCA             ;его с помощью XOR,
26013  C622     ADD   34         ;RRCA, ADD и его
26015  77       LD    (HL),A     ;запись
26016  23       INC   HL         ;следующий адрес
26017  B7       OR    A          ;проверка окончания
26018  ED52     SBC   HL,DE      ;переход в цикл или
26020  19       ADD   HL,DE      ;возврат в БЕЙСИК
26021  20F2     JR    NZ,26009   ;
26022  C9       RET
```

Как видно, она построена аналогично декодирующей процедуре. Если таковой нет, то ее можно просто и быстро написать на основе декодирующей процедуры.

При защите программ применяются также малоизвестные и неопубликованные в фирменных каталогах команды микропроцессора Z80. Благодаря их применению работа программы становится малочитаемой, да и просмотр ее дизассемблером затруднен.

Наиболее часто встречаемыми неопубликованными инструкциями являются команды, оперирующие на половинках индексных регистров IX и IY в группе команд, которым не предшествует никакой иной префикс(т.е. CBH или EDH). Основываются они на префиксации кодом DDH или FDH команды, касающейся регистра H или L. В этом случае вместо этого регистра берется соответствующая половина индексного регистра. Через HX обозначается старшая половина регистра IX, через LX - младшая. Аналогично HY и LY.

Вот примеры:

КОД	КОМАНДА	КОД	КОМАНДА	КОД	КОМАНДА
24	INC H	DD24	INC HX	FD24	INC HY
2D	DEC L	DD2D	DEC LX	FD2D	DEC LY
4C	LD C,H	DD4C	LD C,HX	FD4C	LD C,HY
64	LD H,H	DD64	LD HX,HX	FD64	LD HY,HY
2601	LD H,1	DD2601	LD HX,1	FD2601	LD HY,1
B5	OR L	DDB5	OR LX	FDB5	OR LY

Это верно для всех команд пересылки однобайтовых данных между регистрами восьмибайтовых операций AND, OR, XOR, ADD, ADC, SUB, BC и CP, выполняемых в аккумуляторе.

Префикс FDH или DDH относится ко всем регистрам H, L или HL, присутвующим в команде. Следовательно, в одной инструкции невозможно использование ячейки, адресованной как (HL), регистра HL, H или L одновременно с HX, HY, LX, LY, (в дальнейшем ограничимся регистром IX, но все это относится также к регистру IY), например:

```
66  LD H,(HL)      DD66**  LD HX,(IX+*)
75  LD (HL),L      DD75**  LD (IX+*),IX
65  LD H,L         DD65    LD HX,IX
```

Несколько иначе представляется ротация ячейки, адресуемой индексным регистром, т.е. инструкций, начинающихся кодом DDCB. Инструкция

типа RR (IX+*) и им подобные подробно описаны во всех доступных материалах о микропроцессоре Z80, но мало кто знает об инструкциях типа RR (IX+*),% и им подобным, где % обозначает любой регистр микропроцессора. Они основываются на префиксировании кодом DDH или FDH инструкции типа RR %. Так же обстоит дело с инструкциями SET N,(IX+*),%, а также RES N,(IX+*),% (но для BIT уже нет).. Выполнение такой инструкции основано на выполнении нормальной команды RR (IX+*) (или подобной), SET N,(IX+*) или RES N,(IX+*), а затем пересылки результата как в ячейку (IX+*), так и соответствующий внутренний регистр микропроцессора. Например:

```
CB13        RL E
DDCB0113    RL (IX+1),E
```

вызывает сдвиг ячейки с адресом IX+1 влево на 1 бит и пересылку результата в регистр E, что обычным способом следовало бы сделать так:

```
DDCB0116  RL (IX+1)
DD5E01    LD E,(IX+1)
```

В конце рассмотрения инструкций этого типа следует вспомнить, что команд EX DE,IX или EX DE,IY нет. Префиксование команды EX DE,HL не дает никаких результатов. Также - префиксование команд, коды которых начинаются с EDH, а также тех, в которых не присутствует ни один из регистров H,L или пары HL (например, LD B,N, RRCA и т.д.).

Очередной любопытной командой является SLI (SHIFT LEFT AND INCREMENT), выполнение которой аналогично SLA с той разницей, что самый младший бит устанавливается в 1. Признаки устанавливаются идентично SLA и другим сдвигам:

```
CB37        SLI A
CB36        SLI (HL)
DDCB**36    SLI (IX+*)
DDCB**57    SLI (IX+*),A
```

Временами некоторые проблемы вызывает построение флажкового регистра, особенно тогда, когда он используется достаточно нетипично, например:

```
PUSH      AF
POP       BC
RL        C
JP        NC,...
```

Его вид представлен на рисунке:

7	6	5	4	3	2	1	0
S	Z	F5	H	F3	P/V	N	C

Дополнительной особенностью регистра F является то, что биты 3 и 5 (обозначенные как F3 и F5) точно отражают состояние восьмибитовой арифметической или логической операции, а также команд IN %,(C) и IN F,(C). Например, после выполнения:

```
XOR       A           ;запись значения 0
ADD       A,15        ;результат - 00001111
```

указатели будут: F5 = 0; F3 = 1

Последняя (кто поручится, что их больше нет?) тайна Z80 - это регистр обновления памяти R. А точнее то, что когда после очередного

машинного цикла микропроцессора инкрементируется значение этого регистра, то его старший бит остается неизменным, следовательно, он может быть использован для хранения любой, естественно однобитовой информации. Важно также то, что инструкция LD A,R, с помощью которой может быть получена эта информация, устанавливает также указатель S и, следовательно, не требуется дополнительная инструкция, проверяющая значение этого бита.

В конце - несколько коротких, но часто применяемых мер защиты. Их ликвидация основывается обычно на действиях, обратным защитным. Чаще всего с этой точки зрения эксплуатируются системные переменные. Например, переменная DFSZ (23659) определяет число строк в нижней части экрана, требуемую для вывода сообщения об ошибке или ввода данных. Во время работы программы это значение обычно равно 2, но достаточно занести туда 0, чтобы программа зависла при попытке прерывания (т.к. выводится сообщение, для которого нет места). Если в программе находится инструкция INPUT или CLS, то она имеет подобный результат.

Распространенным способом защиты является также изменение значения переменной BORDCR (23624), которая определяет цвет рамки и атрибутов нижней части экрана. Значение отдельных битов следующее:

7 бит - FLASH
6 бит - BRIGHT
5..3 биты - PAPER
2..0 биты - INK

Способ защиты основывается на том, что цвет чернил - тот же самый, что и у рамки. Следовательно, при остановке программы можно подумать, что она зависла (не видно сообщения). Разблокировать защиту можно, вписав BORDER 0 (или другой цвет).

Очередным простым способом защиты является изменение значения переменной ERRSP (23613, 23614) или дна машинного стека (эта переменная указывает на дно стека), где обычно находится адрес процедуры, обрабатывающей ошибку БЕЙСИКа (вызываемой с помощью RST 8). Уменьшение значения ERRSP на 2 вызывает защиту программы от "BREAK" и любой другой ошибки - программа заново запускается с места, в котором произошла ошибка. Смена содержимого дна машинного стека или другая замена содержимого ERRSP может вызвать зависание или даже рестарт компьютера.

Известным способом защиты является также занесение значений больших 9999 в ячейки памяти, обозначающие номер строки БЕЙСИКа (первый байт - старший). Если он размещается в границах 1000..16383, то на листинге это выглядит, например, 000 (вместо 1000) и строки эти невозможно скорректировать (EDIT). Если же номер превышает 16384, то дальнейшая часть программы считается несуществующей.

Можно также встретить защиту, основанную на занесении минимальных значений (т.е. 1) в переменные REPDEL (23561) и REPPER (23562), что затрудняет работу с компьютером. Для ее ликвидации требуется лишь быстрая реакция (запишите с помощью POKE нормальные значения: 35 и 5).

Стоит еще вспомнить о переменной NMIADD, которая не используется из-за ошибки в ROM. Она должна была содержать адрес подпрограммы обработки NMI-прерывания. Ошибка в том, что микропроцессор переходит туда лишь тогда, когда он равен 0. Если, однако, добавить к SPECTRUM соответствующую приставку с памятью EPROM, то путь к задержке любой программы будет открыт.