

## Материалы к занятию 8 апреля 2020

### Тема: Резидентные программы и их использование.

Резидентная программа – это некоторый (произвольный) код, который остается в памяти после завершения программы. То, что она остается в памяти (резидентно), позволяет использовать этот код из другой прикладной программы. Фактически резидентная программа становится частью операционной системы.

Несколько резидентных программ уже выложены на сайте (они же есть на виртуальном диске для виртуальной машины) в каталоге 20\_04\_15. В этих программах использованы способы защиты от повторной загрузки, описанные в пособии.

Здесь будут рассмотрены два более интересных примера резидентных программ. Первая – структурно простая (она загружается, не проверяя наличие ее копий в памяти, и ее нельзя выгрузить). То, что программа при загрузке не проверяет наличие себя в памяти, позволяет загружать ее многократно. Это приводит к появлению в памяти множества ее копий (и засорению такого ценного ресурса, как оперативная память). Вторая программа выполнена с защитой от повторной загрузки. Кроме того, ее можно выгрузить. Обе программы прилагаются в каталоге 04\_08N в общем архиве. Подробнее об этих программах ниже.

Итак, резидентная программа – это, как уже говорилось выше, произвольный резидентный код, выполняющий что-нибудь полезное. Первая программа, которую мы будем рассматривать (**tsr0.asm**) складывает два четырехзначных десятичных числа, фиксируя при этом возможное переполнение. Расположение резидентного кода схематически показано на рисунке 1.

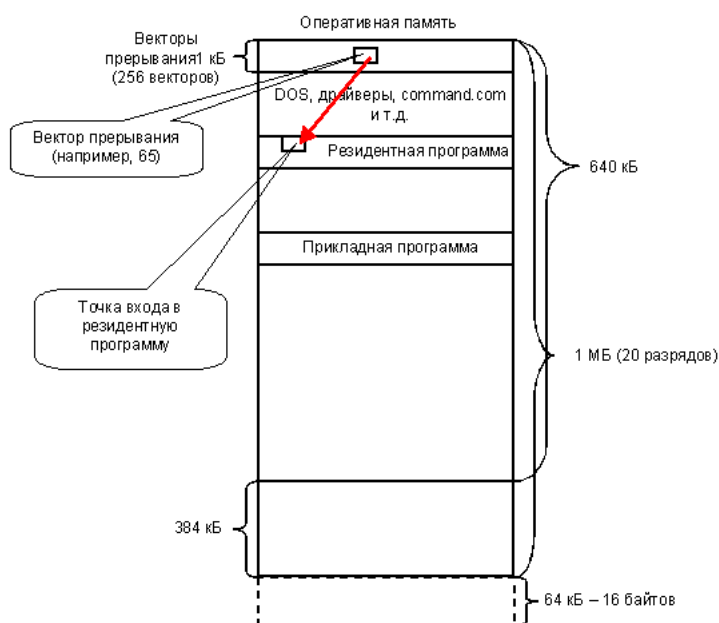


Рисунок 1 – Резидентная программа в памяти

Прежде всего, о том, как оставить резидентный код. Это можно сделать двумя способами:

- функцией **31h int 21h** системного прерывания **int 21h** и
- прерыванием **27h**.

Оба варианта очень схожи. Обе функции называются завершить и оставить резидент (**TSR** – **Terminate and Stay Resident**). От обычных функций завершения прикладной программы они отличаются тем, что при завершении обычной программы память для **DOS** освобождается с начала **PSP** этой программы, а в указанных функциях память освобождается начиная с места, где заканчивается резидентный код, который необходимо оставить в памяти. Для этого в обеих функциях в регистре **DX** указывается размер кода, который должен остаться в памяти. Отличие в том, что в функции **31h int 21h** размер оставляемого кода указывается в параграфах по 16 байт (что позволяет оставить резидентно код любого размера, вплоть до 1 МБ), а в **int 27h** в том же самом **DX** размер оставляемого кода указывается в байтах (что позволяет оставлять не более 64 кБ). Кроме того, в **31h int 21h** задается код возврата в **AL** (при нормальном завершении – это 0), а в **int 27h** код завершения не используется. При этом, оставляя резидентный код необходимо помнить, что с ним остается **PSP** программы, занимающий 256 (100h) байтов.

Когда в памяти остается резидентным код, который может выполнять что-то полезное (в нашей первой программе он может складывать два 4-разрядных десятичных числа), возникает проблема, как к этому коду обратиться. Все дело в том, что когда мы запускаем прикладную программу, которая должна обратиться к резидентному коду, мы не знаем, где именно в памяти он находится. Конечно, можно это определить любой программой просмотра памяти (например, прилагаемой **mi**), но при следующем запуске или на другой машине этот адрес будет другим. Поэтому важно каким-то образом привязать оставляемый резидентный код к заранее известной точке. В пособии описаны различные варианты такой привязки, но наиболее употребимый вариант – использование вектора прерывания (лучше из пользовательского диапазона **60h-67h**). Этот вариант удобен еще и тем, что для дальнего вызова через вектор прерываний нужна 2-байтовая команда (**int n**), в то время, как стандартные дальние команды вызова (**call**) или перехода (**jmp**) – пятибайтовые.

Отметим также, что при запуске резидентная программа является обычной прикладной программой, пока мы не вышли из нее одной из упомянутых команд (**31h int 21h** или **int 27h**). Для того, чтобы подготовить интерфейс связи с оставляемым резидентным кодом нужна секция инициализации, которая делает оставляемый резидентный код удобным для использования. Так как секция инициализации нужна только при подготовке оставления резидента, она обычно располагается в конце программы, и при оставлении резидентного кода не входит в него. Остальные подробности рассмотрим на конкретных программах.

Итак, первая программа **tsr10.asm** приведена ниже.

Напомню, что эта программа не защищена от повторной загрузки, не может быть выгружена и не восстанавливает перехваченные векторы

прерывания. Последний факт связан с тем, что используются пользовательские прерывания **int 65h** и **int 66h** (они не используются **DOS**).

Что должна делать эта программа? Она должна взять из восьмибайтового буфера, адрес которого хранит вектор прерывания **66h** два четырехразрядных десятичных числа. Первое расположено в первых четырех байтах буфера, второе – в последних четырех (5-8). Разряды десятичного числа в буфере расположены следующим образом: от старших разрядов к младшим – от начала к концу (то есть, самый старший десятичный разряд в байте 0, самый младший – в 3). Затем надо сложить их, и поместить результат в буфер второго числа. При сложении должно контролироваться переполнение, и при его возникновении должно выводиться сообщение о переполнении.

Первые строки программы (с 7 по 10) стандартные (может в другом порядке), поэтому комментариев не требуют. Далее в строке 11 идет определение дальней (**far**) процедуры (**resprog proc far**). Процедура обязательно должна быть дальней, так как доступ к резидентному коду будет осуществляться из другой программы, находящейся в произвольном месте оперативной памяти.

Затем идет переход на процедуру инициализации (строка 12 – **jmp init**), которая по уже описанной причине находится в конце программы. Напомню, что в момент запуска этой программы, пока мы из нее не вышли, она является обычной прикладной программой типа **.com**, поэтому во всех сегментных регистрах прописан один и тот же сегментный адрес загрузки программы. Когда мы оставим резидентный код, то управление к нему будет попадать из произвольного места, и на его сегментный адрес будет указывать ТОЛЬКО **CS**. Все остальные сегментные регистры могут иметь совершенно непредсказуемые значения.

Далее, со строки 14 начинается новая процедура обработки прерывания **int 65h**, которое и будет использоваться для вызова резидентного кода.

Далее еще одна особенность любого резидентного кода. Так как управление он получает из чужой программы (то есть, не этой конкретной), то надо позаботиться о том, чтобы при возврате управления в вызывающую программу все регистры имели то же значение, что и при входе в резидентный код. Исключение составляет сегментный регистр **CS**, так как именно с его помощью и происходит вызов резидента. (Разумеется, это касается и указателя команд **IP**, но он непосредственно программно недоступен, поэтому говорить о нем не будем.) Для сохранения всех регистров общего назначения (**POH**) в строке 16 использована команда **pusha**, из-за которой и написано **.286** в строке 9. Напомню, что эта команда сохраняет в стеке регистры **AX, CX, DX, BX, SP, BP, SI, DI**. Если в резидентном коде предполагается «портить» сегментные регистры (кроме **CS**), о них следует позаботиться отдельно.

Так как адрес буфера с числами, которые надо сложить, находится в векторе прерывания **66h**, далее идет получение этого адреса. В строке 17 определяется функция **AH=35h** и номер вектора **AL=66h** (**mov ax,3566h**), а в

строке 18 вызывается системное прерывание (**int 21h**). Напомню, что эта функция возвращает вектор (**сегмент:смещение**) в регистрах **ES:BX**. В принципе, этот адрес обычно сохраняют во внутренних переменных, но мы этого делать не будем, так как портить эти регистры не собираемся.

Затем готовим цикл сложения. Для этого в **SI** в строке 19 (**mov si,3**) определяем номер младшей цифры (у нас младшая цифра находится в третьем байте). Следующую команду надо пояснить (строка 20 – **clc**). При организации многозначного сложения сложения надо младшие разряды складывать командой **add** (без учета флага переноса), а все последующие – командой **adc** (с учетом этого флага, так как в нем фиксируется возможный перенос в следующий разряд). Выделять первую команду сложения из цикла неудобно, поэтому все разряды складываются командой **adc**, а чтобы в результат не вмешалось случайное значение флага переноса, перед циклом флаг переноса сбрасывается в строке 20. В строке 21 инициализируется счетчик цикла сложения (**mov cx,4**).

В строке 23 начинается цикл сложения (**n651: mov al,es:[bx+si]**). Метка (**n651:**) нужна для замыкания цикла, сегментный префикс (**es:**) обязателен, так как именно в регистрах **ES:BX** находится полный адрес буфера с цифрами, индексный регистр (**bx+si**) указывает на текущую цифру (вначале на самую младшую). В итоге в строке 23 в **AL** помещается текущая (младшая) цифра первого числа. В строке 24 (**adc al,es:[bx+si+4]**) с **AL** складывается такая же (регистр **SI**) цифра второго числа (**si+4**) с учетом разряда переноса (**adc**) (напомним, что перед началом цикла мы очистили флаг переноса). В строке 25 выполняется десятичная коррекция полученного результата (**aaa**), так как у нас неупакованные десятичные цифры (по одной в байте). Отметим, что мы не проверяем, являются ли числа, которые мы будем складывать, правильными двоично-десятичными числами. Предполагается, что это будет делать вызывающая программа.

В строке 26 (**mov es:[bx+si+4],al**) полученная цифра результата записывается на место соответствующей цифры второго числа (так договорились), а флаг переноса хранит возможный перенос в старший разряд.

В строке 27 выполняется переход к следующей (старшей) цифре (**dec si** – помним о расположении цифр числа в буфере по старшинству).

Наконец, в строке 28 цикл замыкается (**loop n651**). Переход на указанную метку производится до тех пор, пока в **CX** не нуль (то есть, 4 раза).

Далее в строке 29 проверяется флаг переноса (**jnc n653**). Переход на метку **n653** выполняется в случае, если флаг переноса сброшен (не было переноса). Если флаг переноса установлен, значит произошло переполнение (произошел перенос в пятую цифру), о чем надо сообщить.

Именно это сообщение и формируется в строках с 31 по 37, для чего используется 9 функция системного прерывания.

В строке 31 в **DX** помещается смещение сообщения (**lea dx,msg**). Здесь мы вспомним, что адрес сообщения в девятой функции задается адресом в

**DS:DX.** С сегментным регистром **DS** здесь проблемы, так как он указывает на вызывающую программу. Нам же для выполнения функции 9 нужно указать сегментный адрес сообщения именно в **DS**. Мы так и сделаем, но предварительно сохраним текущее значение **DS** в стеке (чтобы потом его восстановить). Это мы делаем в строке 32 (**push ds**). Затем в строках 33 и 34 мы записываем в **DS** значение из **ES** (**mov ax,cs, movds,ax**). Здесь опять вспомним, что непосредственная пересылка из одного сегментного регистра в другой запрещена.

Затем в строке 35 (**mov ah,9**) задается функция 9, и вызывается системное прерывание (строка 36 – **int 21h**).

Так как **DS** нам теперь не нужен, восстанавливаем его старое значение (строка 37 – **pop ds**).

На следующей строке (строка 38 – **n653: popa**), куда мы попадаем также, если не было переполнения (естественно, без вывода сообщения) восстанавливаются все РОН.

Наконец, в строке 39 происходит выход из процедуры обработки прерывания (**iret**).

В строке 40 находится сообщение о переполнении (**msg db 'Overflow\$'**). Здесь нет переводов строки и возвратов каретки, так как предполагается что этим озаботится вызывающая программа. Так как сообщение это должно выводиться резидентным кодом, оно включено в резидент.

В строке 41 замыкается новая процедура обработки прерывания **65h** (**new\_65 endp**). В следующей строке замыкается главная процедура **resprog** (**resprog endp**).

Так как для оставления резидентного кода необходимо знать его размер, в следующей строке он определяется в байтах (строка 43 – **ressize equ \$-resprog**). Здесь **\$-resprog** определяет расстояние в байтах от начала процедуры **resprog** до текущего места компиляции (**\$**).

В строке 44 начинается процедура инициализации (**init proc near**). Это ближняя процедура, так как выполняется она только при запуске программы.

В строках 45-47 устанавливается новый вектор прерывания **65h** на нашу новую процедуру его обработки (**new\_65**). Старое значение этого векторы мы не сохраняем, так как восстанавливать его не надо (**DOS** его не использует). Итак, строка 45 – **mov ax,2565h** (функция **25h** – установка **65h** вектора прерывания), строка 46 – **lea dx,new\_65** (в **DX** – смещение новой процедуры обработки **65h** прерывания, о **DS** позаботилась **DOS**), строка 47 – **int 21h**, вызов системного прерывания. Таким образом, первое, что мы сделали, это установили точку привязки резидентного кода (вектор прерывания **65h**).

Второе, что надо сделать, это сообщить **DOS** размер оставляемого резидентного кода и выйти из программы функцией **31h**.

Размер оставляемого резидентного кода в параграфах записывается в **DX** в строке 50 (**mov dx,(ressize+10fh)/16**). Здесь деление на 16, чтобы получить размер в параграфах (размер параграфа именно 16 байт), а то, что

делится на 16 состоит из трех частей: **ressize+100h+0fh**. Относительно **ressize** все понятно – это размер полезного резидентного кода в байтах. **100h** – это размер «бесполезного» для нас, но обязательного для **DOS PSP**. (Именно из-за него в начале **.com** программы строит **org 100h**. Последнее слагаемое связано с тем, что размер кода может быть равен не целому числу параграфов. Если не принять специальных мер, при делении на 16 мы получим целое число параграфов, а нецелая часть будет отброшена. Чтобы этого не случилось, мы к размеру в байтах резидентного кода добавляем **0fh**, что при наличии, хотя бы одного байта в нецелой части параграфа, даст еще один целый параграф, и нецелая часть будет учтена. То есть, добавка **0fh** в размере оставляемого кода нужна для округления числа параграфов в большую сторону.

В строке 51 задается **31h** функция системного прерывания (**mov ax,3100h**) – **AH=31h**, и нулевой код возврата **AL=0**, что означает «все в порядке».

В строке 53 замыкается процедура инициализации (**init endp**). Напоминаю, что в резидентный код эта процедура не включается.

Конец программы стандартный – замыкание сегмента кода и обозначение точки входа – пояснений не требует.

Скомпилировав программу стандартным образом, и запустив ее мы получим указанный резидентный код в памяти, что можно посмотреть с помощью программы **mi.com**, находящейся в прилагаемом каталоге. Теперь надо придумать, как этот код использовать.

Для этого мы используем программу **p10.asm**, приведенную ниже. Эта программа должна подготовить два правильных четырехзначных неупакованных десятичных (двоично-десятичных) числа, поместить их в буфер, записать адрес этого буфера в вектор прерывания **66h**, Вызвать резидент командой **int 65h** и распечатать результат. Отметим, что в случае возникновения переполнения сообщение об этом выводит на экран резидентный код. Десятичные числа будем вводить с клавиатуры, проверяя при этом, чтобы они состояли только из цифр 0-9.

Разберем программу. Опять первые четыре строки (4-7) стандартны, и объяснять их не будем. На восьмой строке (**Start: jmp st1**), метка, определяющее точку входа, и перескок через переменные, из которых здесь только одна – буфер для двух чисел (строка 9 – **count db 8 dup(0)**). Как видим, буфер состоит из 8 байтов, заполненных нулями. Нули здесь не просто так. Если вводить будем менее четырех цифр, остальные места должны быть заполнены нулями. При вводе возникает еще одна проблема – вводить числа мы будем естественным образом (сначала старший разряд, затем младшие), а нули должны стоять перед старшими разрядами. Проблему эту мы решим с помощью стека, в котором, как известно, первый вошел – последний вышел. То есть, выходить из стека цифры будут, начиная с младшей цифры (но об этом в соответствующем месте программы).

В строке 10 (**st1: xor di,di**), в которой стоит целевая метка перескока через переменные, инициализируется индекс цифры (**DI**). Так как нам

предстоит вводить два числа, которые должны быть записаны в один и тот же буфер, но со смещением в 4 байта регистр **DI** и будет содержать это смещение (для первой цифры это будет 0, для второй – 4).

В строке 11 (**call in1**) вызывается процедура ввода четырехзначного десятичного числа (универсальная для обоих чисел) со смещением в **DI**, равным нулю. Процедура ввода проверяет, не был ли ввод нулевым, и при нулевом вводе возвращает взведенный флаг переноса. Поэтому в следующей строке (строка 12 – **jc st2**) выполняется условный переход по флагу переноса на вывод сообщения о несостоявшемся вводе, после чего выполняется выход из программы. Если флаг переноса сброшен, переход не выполняется, и мы переходим к следующей строке.

На строке 13 (**mov di,4**) индекс смещения **DI** получает значение 4 для ввода второй строки, а на строке 14 (**call in1**) опять вызывается процедура ввода, теперь уже для второго числа. После этого, в строке 15 (**jc st2**) опять выполняется условный переход по флагу переноса на вывод сообщения о несостоявшемся вводе. Если ввод был ненулевым, можно работать дальше.

Во-первых, надо сообщить резиденту адрес буфера с числами (как мы помним, для этого предназначен вектор прерывания **66h**). Это делается в строках 16-18.

В строке 16 (**mov ax,2566h**) задается 25h функция системного прерывания (**AH=25h**) установки 66h вектора прерывания (**AL=66h**). В строке 17 (**lea dx,count**) в регистр **DX** записывается смещение буфера с цифрами (помним, что о сегментном адресе буфера в **DS** позаботилась **DOS**). Наконец, вызываем системное прерывание (строка 18 – **int 21h**), которое и устанавливает этот вектор прерывания.

Цифры подготовлены, теперь можно вызывать резидент, что мы и делаем в строке 19 (**int 65h**). Как мы помним, резидент складывает полученные два числа, результат помещает на место второго числа и, в случае возникновения переполнения, выводит сообщение на экран.

Так как возможен вывод сообщения резидентом, в первой команде после вызова резидента мы выводим на экран перевод строки и возврат каретки (строка 20 – **call wk**).

После этого, в строках 21 и 22 готовим вывод результата на печать. Для этого в строке 21 (**lea bx,count+4**) записываем в **BX** адрес начала результата (бывшее второе число), не забывая при этом, что по этому адресу находится старший разряд результата. В строке 22 (**mov cx,4**) готовим счетчик печати в **CX** (4 цифры).

В строке 23 начинается цикл печати результата (**st3: mov al,[bx]**). Здесь имеется метка (**st3:**) для замыкания цикла, а в **AL** помещается очередная (сначала старшая) цифра результата.

В строке 24 вызывается процедура печати одной шестнадцатеричной цифры из **AL** (вспомним, что у нас неупакованные десятичные числа, поэтому в байте только одна цифра).

В строке 25 (**inc bx**) переходим к следующей (более младшей) цифре, и в строке 26 (**loop st3**) замыкаем цикл (который будет выполнен **<CX>** раз, то есть, четыре).

После этого ожидаем нажатия клавиши, чтобы увидеть результат (строка 27 – хог **ah,ah** и строка 28 – **int 16h**).

Наконец, в строке 29 (**int 20h**) выходим из программы.

Далее, в строках 31-37 выводится сообщение о несостоявшемся вводе и выполняется выход из программы с ожиданием нажатия клавиши.

В строке 31 (**st2: mov ah,9**) находится метка (**st2:**), на которую выполняется переход в случае нулевого ввода, и готовится функция 9 системного прерывания вывода строки на экран (**AH=9**).

В строке 32 (**lea dx,noin**) в **DX** помещается смещение сообщения (помним о **DS**), а в следующей вызывается системное прерывание (строка 33 – **int 21h**). Четыре последующих строки (34 – **xor ah,ah**, 35 – **int 16h**, 36 – **int 20h**, 37 – **noin db "Не введено число",10,13,'\$'**) пояснений не требуют.

Далее идет вызываемая процедура ввода десятичного числа (максимально четырехзначного). Процедура оформлена именно в виде процедуры (строка 39 – **in1 proc near**, строка 68 – **in1 endp**).

В строке 40 инициализируется счетчик максимального числа вводимых цифр (**mov cx,4**), а в строке 41 (**mov si,0**) – счетчик действительно введенных цифр. Если вводятся все 4 цифры, то ввод автоматически заканчивается. Если же надо ввести меньшее число цифр, ввод завершается нажатием клавиши **Enter** (при этом нам надо знать, сколько именно цифр было введено, чтобы обеспечить ведущие нули в буфере).

В строке 42 (**in1: xor ah,ah**) начинается цикл ввода цифр (метка **in1:** для замыкания цикла и нулевая функция клавиатурного прерывания (строка 43 – **int 16h**). Прежде всего (строка 44 – **cmp al,0dh**) введенный символ проверяется на символ возврата каретки (**0dh=13**). Если введен этот символ, значит ввод числа окончен до ввода 4 цифр. В этом случае (строка 45 – **jz in12**) выполняется условный переход на метку **in12**.

Если был введен не возврат каретки, проверяется, введена ли цифра от 0 до 9. Для этого в строке 46 (**cmp al,30h**) содержимое **AL** сравнивается с кодом **30h** (код цифры нуль). Если введенный символ имеет код, меньший, чем **30h**, значит введена не цифра. В этом случае (строка 47 – **jc in11**) выполняется условный переход (по флагу переноса) на начало цикла ввод, то есть, символ просто игнорируется.

Похожая проверка выполняется на строке 48 (**cmp al,3ah**) с кодом **3ah**. Код цифры должен быть меньше этого значения. Если условие не выполняется, на строке 49 (**jnc in11**) выполняется условный переход (по флагу переноса) на начало цикла ввод, то есть, символ опять игнорируется.

Если переход не выполняется, значит введена цифра от 0 до 9 (то есть, введено правильное двоично-десятичное число). В этом случае введенная цифра (в **AL**) выводится на экран вызовом процедуры печати одной цифры (строка 61 – **call prsn**). Затем в **AL** код цифры преобразуется в собственно цифру (строка 52 – **sub al,30h**), путем вычитания из кода значения **30h**



(напомню, что коды цифр от 0 до 9 соответствуют ASCII кодам от 30h до 39h).

В строке 53 (**push ax**) полученное значение помещается в стек ( в том числе, и для инверсии порядка цифр).

Затем (строка 54 – **inc si**) счетчик количества реально введенных цифр увеличивается на единицу.

Строка 55 (**loop in11**) замыкает цикл ввода.

На следующую 56 строку мы можем попасть, либо после естественного завершения цикла (когда в **CX** нуль, и введены 4 цифры), либо после нажатия клавиши **Enter** (ввода символа возврата каретки). В этом случае надо проверить, была ли введена хотя бы одна цифра. Для этого здесь использован несколько экзотический способ: значение из **SI** пересылается в **CX** (строка 56 – **in12: mov cx,si**), а затем выполняется условный переход (по условию равенства нулю содержимого **CX** на метку **in14** (строка 57 – **jcxz in14**). То есть, переход на эту метку произойдет лишь в случае ввода нуля цифр. (Можно было сделать это вполне традиционно, например, строка 56 – **and si,si** (или **or si,si**), а затем в строке 57 **jz in14**, эффект был бы тот же).

Если переход не был выполнен, значит был ввод от 1 до 4 цифр (и все они были помещены в стек, причем младший из введенных разрядов находится в вершине стека), поэтому в строке 58 (**call wk**) выполняется переход на новую строку, чтобы отделить последующий вывод от отображенного числа.

После этого производится запись введенного числа в буфер.

В строке 59 (**in13: pop ax**) из стека извлекается младший (последний записанный) разряд числа (действителен только **AL**). Метка **in13:** нужна для замыкания цикла записи в буфер.

В строке 60 (**lea bx,count**) в **BX** помещается смещение буфера чисел. Затем в строке 61 (**mov [bx+di+3],al**) извлеченная цифра записывается в буфер в самый старший байт числа (первого, если **DI=0**, или второго, если **DI=4**).

Далее к строке 62 (**dec di**) выполняется переход к следующему (более младшему) разряду в буфере, а в строке 63 (**loop in13**) замыкается цикл записи. Напомню, что цикл записи повторяется столько раз, какое значение в **CX**, а в **CX** значение, взятое из **SI** – количество фактически введенных цифр. То есть, в буфер будут записаны введенные цифры, и, если их было меньше 4, то в оставшихся старших разрядах останутся нули, которые находились там изначально.

В строке 64 сбрасывается флаг переноса (**clc**), являющийся признаком успешного ввода. Затем, в строке 67 (**ret**) выполняется выход из процедуры.

Как уже говорилось ранее, в строке 68 замыкается процедура (**in1 endp**).

Ниже расположены процедуры печати одной hex цифры из **AL**, печати 1 ASCII символа, печати перевода строки и возврата каретки и печати пробела, которые либо уже неоднократно рассматривались, либо совершенно

очевидны. Поэтому рассматривать их здесь не будем. (Если, все-таки, возникнут вопросы, спрашивайте.)

Возможные эксперименты с программой:

- после загрузки программы посмотрите программой **mi.com** ее место в памяти,
- убедитесь в том, что при многократном запуске программы в памяти появляются ее копии,
- попробуйте различные сочетания чисел с различным числом цифр, в возможными переносами между разрядами, с возможным переполнением, с нулевым вводом, наконец,
- попробуйте изменить максимальное количество цифр, например, на 5 (это существенно сложнее),
- попробуйте выводить сообщение о переполнении кириллицей,
- попробуйте придумать что-нибудь еще.

### Программа tsr10.asm

```
; 2018 TSR сложения и печати четырехзначных десятичных чисел
; int 65h - вход в сложение,
; int 66h - адрес буфера для чисел
; первые 4 байта - первое десятичное число
; вторые 4 байта - второе десятичное число
; Результат пишется в буфер второго числа
Assume CS: Code, DS: Code
Code    SEGMENT
        .286
        org 100h
resprog proc    far    ;Дальняя, т.к.резидентная
        jmp  init      ;Переход на инициализацию
; Новый обработчик прерывания 65h
new_65  proc     far    ;См. выше
; Получение адреса буфера чисел
        pusha
        mov  ax,3566h ;Функция взятия вектора 66
        int  21h      ;возвращает в регистрах ES:BX
        mov  si,3 ;Номер цифры
        cld  ;Сброс флага переноса (для цикла сложения)
        mov  cx,4 ;Счетчик цифр
; Цикл сложения
n651:    mov  al,es:[bx+si] ;Очередная цифра первого числа
        (младшая)
        adc  al,es:[bx+si+4] ;Сложение с цифрой второго числа
        aaa      ;Десятичная коррекция
        mov  es:[bx+si+4],al ;Запись результата
        dec  si   ;Переход к следующей цифре
        loop n651 ;На начало цикла
        jnc  n653 ;Переход, если не был перенос
; Здесь был перенос, значит переполнение
n652:    lea  dx,msg      ;Сообщение о переполнении
        push ds          ;Сохранение DS
        mov  ax,cs       ;Так как резидент, обязательно
```

```

        mov  ds,ax      ;DS=CS
        mov  ah,9 ;Функция вывода строки
        int  21h ;Системное прерывание
        pop  ds ;Восстановление DS
n653:   popa
        iredt
msg     db   'Overflow$'
new_65  endp
resprog endp
ressize equ  $-resprog ; Размер в байтах резидентной части
init proc near
        mov  ax,2565h ; Функция 25h, вектор 65h
        lea  dx,new_65
        int  21h      ; Запись нового вектора 65h
; Завершение программы инициализации с оставлением
; резидентной части в памяти
        mov  dx,(ressize+10fh)/16 ; Указание размера рез кода
        mov  ax,3100h ; Завершить и оставить резидентной
        int  21h
init endp
Code    ENDS
        END  resprog

```

### Программа вызова резидента tsr10

```

; Программа вызова tsr10
; Адрес вызова вектор 65h
; Указатель на данные вектор 66h
Code SEGMENT
.286
        Assume CS: Code, DS: Code
        org  100h
Start:   jmp  st1
count    db   8 dup(0);Буфер для двух чисел
st1:     xor  di,di      ;Смещение для первого числа
        call in1 ;Ввод первого числа
        jc  st2 ;Переход, если нулевой ввод (FC=1)
        mov di,4 ;Смещение для второго числа
        call in1 ;Ввод второго числа
        jc  st2 ;Переход, если нулевой ввод
        mov ax,2566h ;Установка 66 вектора прерывания
        lea dx,count ;Смещение буфера (DS)
        int 21h ;66 вектор - адрес буфера
        int 65h ;Вызов резидента
        call wk ;ВК (если было переполнение)
        lea bx,count+4 ;Адрес начала результата
        mov cx,4 ;Количество выводимых цифр
st3:     mov  al,[bx] ;Очередная (первая) цифра
        call prsn ;Печать очередной цифры
        inc  bx ;Переход к следующей цифре
        loop st3 ;Замыкание цикла
        xor  ah,ah ;Функция 0
        int 16h ;Клавиатурного прерывания
        int 20h ;Выход из программы

```

```

;Зесь не было ввода цифр (предупр. и выход)
st2: mov  ah,9 ;Функция 9 (вывод строки)
      lea  dx,noin ;Адрес строки
      int  21h ;Системное прерывание
      xor  ah,ah ;Функция 0
      int  16h ;клавиатурного прерывания
      int  20h ; Выход из программы
noin db  "Не введено число",10,13,'$'
; Процедура ввода цифр (до 4) с клавиатуры
in1  proc near
      mov  cx,4 ;Количество цифр
      mov  si,0 ;Подсчет числа цифр
in11: xor  ah,ah ;Функция 0 (с ожиданием)
      int  16h ;Клавиатурное прерывание
      cmp  al,0dh ;Enter?
      jz   in12 ;Ввод закончен
      cmp  al,30h ;Проверка <0
      jc   in11 ;Не цифра (игнорируем)
      cmp  al,3ah ;Проверка >9
      jnc  in11 ;Не цифра (игнорируем)
; Здесь цифра от 0 до 9
      call prsn ;Показ на экране введенной цифры
      sub  al,30h ;Перевод сивола в цифру
      push ax ;Введенную цифру в стек
      inc  si ;Подсчет введенных цифр
      loop in11 ;На начало цикла
in12: mov  cx,si ;Количество введенных цифр
      jcxz in14 ;Переход, если CX равен нулю
      call wk ;ВК
in13: pop  ax ;Последняя (младш.) цифра из стека
      lea  bx,count ;Смещение буфера
      mov  [bx+di+3],al ;Запись в буфер с младшей цифры
      dec  di ;Переход к след.(старшей) цифре
      loop in13 ;На начало цикла
      cld ;CF=0 - Признак успешного ввода
      ret
in14: stc ;CF=1 - признак отсутствия ввода
      ret
in1  endp
;*****
; Печать одного hex символа из мл. тетр. al
prsn: pusha ;Сохранение всех регистров общего назначения
      and  al,0fh ;Выделение младшей тетрады
      add  al,30h ;Перевод цифры в ее код (0-9)
      cmp  al,39h ;Проверка >9
      jle  pr1 ;Переход, если >9
      add  al,7 ;Перевод цифры в код (A-F)
; Печать 1 ASCII символа
pr1: mov  bx,0fh ;Цвета (фон 0-черный, чернила f-белый)
      mov  ah,0eh ;Вывод симв. в режиме телетайпа
      int  10h ;Видеоперывание
      popa ;Восстановление всех РОН
      ret ;Выход

```

```

print:    pusha
          jmp prl ;На печать 1 ASCII символа
wk:       pusha
          mov al,0ah ;Перевод строки
          call print ;Печать
          mov al,0dh ;Возврат каретки
          call print ;Печать
          popa
          ret
space:    pusha
          mov al,32 ; Код пробела
          call print ; Печать 1 ASCII символа
          popa
          ret
Code      ENDS ;Конец сегмента
          END Start ;Точка входа

```

Теперь рассмотрим другую программу, которая, с одной стороны, функционально существенно проще (она лишь заменяет при вводе с клавиатуры символ восклицательного знака ! (код 21h) символом решетки # (код 23h). Зато организационно эта программа сложнее:

- она защищена от повторной загрузки,
- она может быть выгружена.

Стоит отметить, что, в отличие от методов, использованных в программах **tim3.asm** и **xchg.asm** из каталога 20\_04\_15, а также методов, описанных в пособии (и в большинстве книг по ассемблеру) здесь использован абсолютно надежный способ защиты от повторной загрузки – проверка наличия в памяти собственного резидентного кода.

Для выгрузки программы предусмотрен ключ «/u», что, может быть недостаточно, так как, наверное, стоило бы предусмотреть еще и ключи «/U», «-u», «-U». Однако, так как программа является учебным примером, обойдемся единственным ключом.

Итак, программа **tsr0.asm**.

Первые 6 строк (3-8) полностью совпадают с первыми шестью строками предыдущей программы. Описывать не будем.

Далее идут переменные – их три. Две переменные для сохранения полного адреса старой процедуры обработки прерывания **int 16h** (строка 10 – **old\_int16\_off dw ?**, строка 11 – **old\_int16\_seg dw ?**) и переменная для хранения сегментного адреса резидента (необходим для выгрузки резидента) **adr\_psp dw ?** в строке 12. Все три переменные не инициализированы (то есть, им не присвоены значения), для них просто зарезервировано место (с метками).

Далее идент новый обработчик прерывания **int 16h** (строка 14 – **new\_int16 proc far**) оформленный в виде дальней процедуры (причины те же, что и выше). Остановимся на функциях этой процедуры. Необходимо отловить запрос перехваченного прерывания с функциями **0** и **10h** (обе функции – ввод с клавиатуры с ожиданием нажатия), для этих функций

выяснить, введен ли символ «!» (код **21h**) и, если это так, заменить его на символ решетки «#» (код **23h**). Если был введен другой символ, надо просто передать управление старой процедуре обработки этого прерывания.

В строке 15 проверяется, не запрошена ли функция 0 (**cmp ah,0** – **AH=0?**). Если это она, условный (по флагу нуля) переход на метку **new3** (**jz new3**). В строке 16 проверяется, не запрошена ли функция **10h** (**cmp ah,10h** – **AH=10h?**). Если это не она, условный (по флагу не нуля) переход на метку **new1** (**jnz new1**).

Если это все же функция **10h**, переход не выполняется, и мы попадаем на метку **new3** (то есть, и для функции **0**, и для функции **10h** мы попадаем на эту метку). Здесь мы должны вызвать старую процедуру обработки прерывания **int 16h** (чтобы не писать самим все ее функции). Проблема, однако, состоит в том, что командой **int 16h** мы ее вызвать не можем, так как ее вектор мы перехватили, и на эту команду будет опять вызвана наша процедура и т. д. Следовательно, ее надо вызывать командой **call**, которая отличается от команды **int** отсутствием действий с регистром флагов (команда **call** не сохраняет в стеке регистр флагов). Однако, старая процедура обработки **int 16h** (как и все процедуры обработки прерываний) завершается командой **iret**, которая извлекает регистр флагов из стека. Чтобы не было нарушения синхронизации стека, мы перед использованием команды **call** для старой процедуры должны сохранить в стеке регистр флагов, что и делается в строке 19 (**new3: pushf**).

Затем в строке 20 мы вызываем старую процедуру (**call dword ptr cs:old\_int16\_off**). Вспомним, что эта команда производит вызов процедуры по адресу, записанному в порядке смещение, сегмент (4 байта), начиная с указанного адреса. В качестве адреса у нас указана переменная для хранения смещения старой процедуры обработки **int16** (а за ней идет переменная с сегментом ее же). Записан адрес старой процедуры обработки производился в секции инициализации в момент установки резидента. Отработав, старая процедура вернет нам в **AL** введенный **ASCII** код (остальное нас не волнует).

В строке 21 (**cmp al,21h**) выполняется проверка введенного символа на «!» (код **21h**). Если введен другой символ, выполняется условный переход (по флагу нуля) на метку **new2** (строка 22 – **jnz new2**) с последующим выходом из процедуры.

Если же был введен символ «!», переход не выполняется, и в строке 22 (**mov al,23h**) символ в **AL** заменяется на решетку (# – код **23h**), после чего выполняется выход из процедуры (строка 24 – **new2: iret**).

В строке 25 находится метка **new1**, на которую осуществляется переход при других функциях клавиатурного прерывания (со строки 18). В этой строке выполняется безусловный переход на старую процедуру обработки прерывания **int 16h** (**new1: jmp dword ptr cs:old\_int16\_off**). Адресация выполняется так же, как и в строке 20. Конечно, можно было бы выполнить команды, как в строках 19 и 20 с последующим выполнением **iret**, однако использованный в строке 25 вариант при том же эффекте более экономный.

В строке 26 замыкается новая процедура (**new\_int16 endp**), а в следующей строке (строка 27 – **resprog endp**) – процедура **resprog**.

В строке 28 уже описанным способом определяется размер оставляемого резидента (**ressize equ \$-resprog**).

В этой программе дополнительно определяется еще размер собственно резидентной процедуры, без переменных (**rs equ \$-new\_int16**) для проверки ее наличия в памяти. Переменные нельзя включать в опознаваемый код, так как они будут отличаться от оригинала, и программа опознана не будет.

В строке 30 начинается процедура инициализации (**init proc near**). Процедура ближняя, так как она выполняется только при запуске программы.

Первым делом проверяется, не был ли введен ключ выгрузки (**/u**).

Как известно, командная строка помещается в **PSP**, начиная с адреса **81h**, а по адресу **80h** находится количество введенных символов командной строки. Из сказанного ясно, что длина командной строки не может превышать 127 символов.

В строке 32 в **BX** загружается адрес **80h** (количество введенных символов командной строки) – **mov bx,80h**. Затем содержимое этой ячейки памяти загружается в **CL**, так как это лишь 1 байт (строка 33 – **mov cl,[bx]**). Для получения полноценного (16-разрядного) счетчика обнуляется старший байт регистра **CX** (строка 34 – **xor ch,ch**).

В строке 35 выполняется команда **jcxz (jcxz cmd2)**, которая выполняет переход на метку **cmd2**, если в **CX** нуль. Эта команда удобнее обычного условного перехода, так как перед ней (в отличие от обычного условного перехода) не надо ничего проверять. Таким образом, если командной строки не было (**CX=0**), то выполняется переход на метку **cmd2**.

Если в **CX** не нуль, содержимое **BX** увеличивается на 1 (переходим к адресу первого символа командной строки) – строка 36 – **inc bx**.

Со строки 37 начинается цикл расшифровки командной строки

В строке 37 текущий (сначала первый) символ командной строки помещается в **AL** (**cmd: mov al,[bx]**). Метка в этой команде нужна для замыкания цикла.

Первым делом текущий символ проверяется на пробел (код 20h). Его надо игнорировать (это правило хорошего тона при обработке командной строки. Это строка 38 (**cmp al,20h**). Если это пробел, переходим на метку **cmd1** (строка 39 – **jz cmd1**). Если это был не пробел, переход в строке 39 не выполняется, и мы переходим к строке 40 (**cmp al,'/'**), где символ проверяется на слэш (/) – признак ключа. Если это не слэш, опять переходим на метку **cmd2** (как и в строке 35) – строка 41 – **jnz cmd2**.

Если, все же, это был слэш, переход в строке 41 не выполнен, мы попадаем на строку 42 (**cmp byte ptr [bx+1],'u'**), где проверяем следующий за слэшем символ на букву **u** (здесь пробелы не допускаются).

Если проверка дает отрицательный результат (не **u**), опять идем на метку **cmd2** (строка 43 – **jnz cmd2**). Если переход не выполнен, значит это была буква **u**, и, следовательно, мы получили ключ выгрузки (**/u**), следствием

чего должна быть выгрузка программы (точнее, освобождение занимаемого резидентом блока памяти).

Выгрузка резидента начинается в строке 46 (**call loaded**), где проверяется, есть ли что выгружать, то есть, присутствует ли наш резидент в памяти. Процедура **loaded** проверяет это, и возвращает взведенный флаг переноса, если резидент в памяти. В противном случае флаг переноса сброшен.

В строке 47 (**jc uninst**) выполняется условный переход (по флагу переноса) на собственно выгрузку (так как есть, что выгружать).

Если переход в строке 47 не выполнен, значит резидента в памяти нет, о чем в строках с 48 по 50 выводится сообщение (**lea dx,msgno, mov ah,9, int 21h**). Вывод сообщения как обычно использует функцию 9 системного прерывания, а сообщение – **msgno**.

После вывода сообщения выполняется выход из программы (строка 51 – **int 20h**), так как выгружать нечего, а от нас больше ничего и не требуется.

В строке 52 (**uninst: call set\_int**) находится метка **uninst:**, на которую мы приходим, когда был ключ **/u**, и есть, что выгружать. Для этого вызывается процедура **call set\_int**, которая и выгружает резидента, предварительно восстановив перехваченный вектор прерывания **int 16h**. После выгрузки резидента происходит выход из программы (строка 53 – **int 20h**), так как мы сделали все, что было надо.

На строке 54 находится команда с меткой **cmd1: (cmd1: inc bx)**, на которую выполняется переход при обнаружении пробела, который должен быть проигнорирован. Эта команда формирует в **BX** адрес следующего символа командной строки, после чего в строке 55 (**loop cmd**) цикл замыкается. Напомню, что цикл будет выполняться столько раз, сколько символов было в командной строке (либо меньше, если ранее встретился ключ выгрузки или другой ключ – не **/u** – или любой символ, отличающийся от пробела или от слэша – **«/»**).

На строке 56 находится метка **cmd2 (cmd2:)**. Так как в этой строке нет команды, то метка относится к команде на следующей строке. На эту метку мы попадаем в том случае, когда в командной строке встретился любой, отличный от пробела, необрабатываемый символ, либо, когда был введен другой ключ (любой символ после слэша). В этом случае считается, что командной строки вообще не было, значит, нас просили загрузить резидент.

Перед загрузкой резидента опять стоит выяснить, не находится ли он уже в памяти. Для этого в строке 58 (**call loaded**) вызывается процедура, проверяющая это, и возвращающая взведенный флаг переноса, если резидент в памяти.

Если резидента нет (флаг переноса сброшен), команда **jnc first\_start** в следующей (59) строке передает управление на метку **first\_start**, с которой начинается собственно установка резидента. Если резидент в памяти, переход в строке 59 не выполняется, и программа переходит к выводу сообщения о том, что программа уже в памяти (строка 60 – **lea dx,msg**, строка 61 – **mov ah,9**, строка 62 – **int 21h**). После этого выполняется выход из



программы (строка 63 – **int 20h**). Все эти действия уже описывались неоднократно.

В строке 64 находится метка **first start:**, опять в одиночестве. После нее начинается собственно установка резидента.

Сначала берется и сохраняется в переменных старый вектор прерывания **int 16h**: строка 66 – **mov ax,3516h**, строка 67 – **mov cs:old\_int16\_off,bx**, строка 68 – **mov cs:old\_int16\_seg,es**. Эти действия также уже обсуждались. Отметим лишь, что здесь префикс **cs:**, который при этих переменных был в процедуре обработки прерывания, здесь не нужен, так как в момент выполнения процедуры инициализации это простая прикладная программа (а не резидент).

Затем устанавливается новый вектор **int 16h**: строка 71 – **lea dx,new\_int16**, строка 72 – **mov ax,2516h**, строка 73 – **int 21h**. Эти действия также неоднократно описывались.

Завершением процедуры инициализации является сообщение **DOS** размера оставляемого резидента: строка 76 – **mov dx,(ressize+10fh)/16**, строка 77 – **mov ax,3100h**, строка 78 – **int 21h**. Все это тоже известно.

Строкой 79 (**init endp**) замыкается процедура инициализации.

Со строки 81 (**loaded proc near**) начинается процедура проверки наличия резидента в памяти. Процедура также ближняя по той же причине, что и инициализация.

Проверка наличия резидента в памяти заключается в сравнении кода новой процедуры обработки прерывания **int 16h** в теле программы и в предполагаемом месте памяти. Если резидент находится в памяти, то сегмент вектора прерывания **16h** должен указывать на начало **PSP**, сопровождающего резидент, а, начиная со смещения метки **new\_int16** в резиденте должна быть расположена такая же процедура, конечно, в случае, если резидент находится в памяти. Если резидента в памяти нет, то по указанным адресам будут находиться совершенно другие коды.

В соответствии со сказанным выше, проверка начинается со взятия **16h** вектора прерывания (точнее, его сегмента). Это строки 82 (**mov ax,3516h**) и 83 (**int 21h**). Насколько мы помним, функция **35h** системного прерывания возвращает сегментный адрес вектора (который нам нужен) в регистре **ES**.

В строке 84 (**mov adr\_psp,es**) полученный сегментный адрес предполагаемого резидента записывается в переменную **adr\_psp**. Для сравнения двух последовательностей кодов мы будем использовать строковую команду **cmpsb**, выполняющую байтовое сравнение элементов памяти **ES:DI** и **DS:SI**. Пара **ES:DI** будет относиться к предполагаемому резиденту, а пара **DS:SI** – к местной процедуре. После строки 84 в сегментные регистры **ES** и **DS** уже загружены нужные сегментные адреса (в **ES** – командой в строке 84, в **DS** – сам же **DOS** при загрузке программы). Осталось загрузить регистры **DI** и **SI**, причем их содержимое (смещение процедуры **new\_int16** относительно начала сегмента) одинаково в резиденте и в местной программе.

В строке 86 (**lea di,new\_int16**) в **DI** загружается это смещение, а в строке 87 (**mov si,di**) – оно же в **SI**. Итак, адреса областей сравнения подготовлены. Осталось инициализировать счетчик сравнения, что и делается в строке 87 (**mov cx,rs**). Неплохо было бы сбросить флаг направления (командой **cld**), однако мы воспользуемся тем, что при загрузке программы **DOS** очищает все регистры. Наконец, строка 88 (**repz cmpsb**) – собственно сравнение. Префикс повторения обеспечивает повторение команды до тех пор, пока есть совпадение (взведен флаг нуля).

Сразу после этой команды, в строке 89, строит условный переход (**jcxz lo1**) по нулевому содержимому **CX** на метку **lo1**. Этот переход будет выполнен лишь в том случае, когда сравнение успешно прошло до конца (то есть, до **CX=0**). А если сравнение успешно завершилось, значит коды совпадают, и, следовательно, резидент находится в исследуемой области памяти).

Если переход не выполнен, значит **CX** не равен нулю, и сравнение не дошло до конца (то есть, возникло несовпадение). Это значит, что коды не совпадают, а, следовательно, резидента в памяти нет. О чем и свидетельствует сброшенный в строке 90 (**clc**) флаг переноса. После этого осуществляется возврат из процедуры (строка 91 – **retn**).

Если переход в строке 89 был выполнен (когда сравнение прошло успешно), в строке 92 (**lo1: stc**) взводится флаг переноса, что свидетельствует о наличии резидента в памяти. Затем в строке 93 (**retn**) осуществляется возврат из процедуры.

Последняя процедура – восстановление вектора прерывания и освобождение блока памяти (выгрузка резидента).

Процедурные скобки – строки 95 – **set\_int proc near** (начало процедуры) и 112 – **set\_int endp** (конец процедуры).

Тело процедуры начинается в строке 97 (**push ds**), где **DS** сохраняется в стеке, так как этот регистр нужен для функции **25h** (напомним, что для записи вектора прерывания необходимо полный адрес записать в пару регистров **DS:DX**).

В строке 98 (**mov dx,es:old\_int16\_off**) в **DX** записывается смещение, а в строке 99 (**mov ds,es:old\_int16\_seg**) в **DS** – сегмент старой процедуры обработки прерывания **int 16h**.

В строке 100 выбирается функция **25h (AH=25h)** установки вектора прерывания **16h (AL=16h)** – **mov ax,2516h**, а в строке 101 вызывается само системное прерывание (**int 21h**). Затем в строке 102 восстанавливается прежнее значение в сегментном регистре **DS (pop ds)**.

Далее освобождается занятый резидентом блок памяти, сегментный адрес которого у нас хранится в переменной **adr\_psp**. Для этого в строке 104 этот адрес помещается в регистр **ES** (сегментный адрес освобождаемого блока памяти), а в строке 104 задается функция **49h** освобождения блока памяти (**mov ah,49h**) системного прерывания и, наконец, вызывается само системное прерывание (строка 106 – **int 21h**). Все, блок памяти освобожден.

Теперь надо вывести на экран сообщение о том, что программа выгружена: строка 107 – **mov ah,9**, строка 108 – **lea dx,un**, строка 109 – **int 21h**).

В строке 110 осуществляется выход из процедуры инициализации (**retn**).

Далее, в строках 112-114 расположены три сообщения, который выводятся программой, после чего в строках 115 и 116 идет стандартнок завершение программы.

Эксперименты с программой:

загрузите резидент, просто запустив программу **tsr0.com**, а затем убедитесь с помощью программы **mi.com**, что **tsr0** действительно в памяти;

попробуйте еще раз загрузить резидент, посмотрите сообщение, которое выдает программа (для этого надо скрыть панели нортон клавишами **Ctrl+o**; для обратного включения панелей надо опять нажать **Ctrl+o**);

попробуйте выгрузить программу, задав ключ **/u**; с помощью программы **mi.com** убедитесь, что программы **tsr0** больше нет в памяти,

попробуйте изменить заменяемый символ, например, на букву **Ы**,

попробуйте изменить символ замены, например, на символ **\$**.

Если что непонятно, спрашивайте.

### Программа **tsr0.asm**

```
; Программа, заменяющая символ !-код 21h решеткой (# - код 23h)
; Выгрузка - /u
Assume CS: Code, DS: Code
Code SEGMENT
.286
    org 100h
resprog proc far
    jmp init
; Сохранение старых векторов прерывания
old_int16_off dw ? ;Смещение
old_int16_seg dw ? ;сегмент вект. 16h
adr_psp dw ? ;Сегм.адр.резидента
; Новый обработчик прерывания 16h
new_int16 proc far
    cmp ah,0 ;Функция 0 - ввод с ожиданием нажатия
    jz new3 ;Она
    cmp ah,10h ;Еще функц. с ожиданием нажатия
    jnz new1 ;Другая функция - надо уходить
new3:    pushf ; Синхронизация стека (из-за iret)
    call dword ptr cs:old_int16_off; На старый обраб.
    cmp al,21h ; Введен «!» ?
    jnz new2 ; нет
    mov al,23h ; Замена на код решетки
new2:    iret ; Выход (введен другой символ)
new1:    jmp dword ptr cs:old_int16_off; На стар. (др.функ.)
new_int16 endp
resprog endp
```

```

ressize equ    $-resprog ;Размер в байтах резидентной части
rs    equ    $-new_int16 ;Размер проверяемого куска (без переменных)
init proc near
; Проверка ключа выгрузки
    mov    bx,80h        ;Счетчик командной строки
    mov    cl,[bx]        ;кол-во симв. в CX
    xor    ch,ch        ;
    jcxz   cmd2 ;
    inc    bx    ;Начало командной строки
cmd:  mov    al,[bx] ;Символ в AL
    cmp    al,20h    ;Пробел?
    jz     cmd1      ;да - игнорировать
    cmp    al,'/'    ;Признак ключа
    jnz    cmd2      ;Не ключ - на загрузку
    cmp    byte ptr [bx+1],'u' ;Выгрузить?
    jnz    cmd2      ;Не u - на загрузку
; Освобождение блока памяти
; Проверка загруженности перед выгрузкой
    call   loaded      ;fc=1 - в памяти
    jc     uninstd     ;Программа в памяти
    lea    dx,msgno    ;Вывод сообщения, что
    mov    ah,9        ; выгружать нечего
    int    21h
    int    20h
uninst: call    set_int ;Восст.вект.прер. и осв.бл.пам.
    int    20h
cmd1:    inc    bx    ;К след.симв.командн.строки
    loop   cmd    ;На начало анализа ком. строки
cmd2:    ;Сюда приходим, когда нет ключа
; Проверка загруженности перед установкой
    call   loaded      ;fc=1 - в памяти
    jnc    first_start ;Не установлена
    lea    dx,msg      ;Вывод сообщения, что программа
    mov    ah,9        ;уже в памяти
    int    21h
    int    20h ;Выход, так как программа в памяти
first_start:
; Перехват 16h с сохранением старого
    mov    ax,3516h ;Взятие старого вектора прерывания 16h
    int    21h
    mov    old_int16_off,bx ;Смещ.стар.вект.16h
    mov    old_int16_seg,es ;Сегм.стар.вект.16h
; Установка нового 16h
    lea    dx,new_int16 ;Смещ.нов.процед. 16h
    mov    ax,2516h      ;Сегм.нов.процед. 16h
    int    21h
; Завершение программы инициализации с оставлением
; резидентной части в памяти
    mov    dx,(ressize+10fh)/16 ;Размер резид. в парагр.
    mov    ax,3100h ;Функция TSR с кодом возврата 0
    int    21h
init endp
; Проверка наличия резидента в памяти

```

```

loaded    proc near ;Проверка загруженности: fc=1 - в памяти
    mov    ax,3516h ;Взятие вектора
    int     21h
    mov    adr_psp,es;Сохранение сегм.адр.резидента
    lea    di,new_int16;ES:DI рез.адрес сканирования
    mov    si,di      ;DS:SI Здешний адрес сканирования
    mov    cx,rs       ;Размер области сканирования
    repz   cmpsb       ;Сравнение, пока совпадает (ZF=1)
    jcxz   lol        ;Если CX=0, значит совпало все
    cld      ; Сброс флага переноса = нет в памяти
    retn
lol: stc      ; Установка флага переноса = в памяти
    retn
loaded    endp
set_int    proc near
; Восстановление старого вектора 16h
    push    ds ; Сохранение DS
    mov     dx,es:old_int16_off ; В DX смещ.из резидента
    mov     ds,es:old_int16_seg ; В DS сегм.из резидента
    mov     ax,2516h ; Установка старого вектора 16h
    int     21h
    pop     ds ; Восстановление DS
; Освобождение блока памяти резидента
    mov     es,adr_psp ; Сегм.адр.резидента
    mov     ah,49h ; Освобождение памяти
    int     21h
    mov     ah,9 ;Функция вывода строки
    lea     dx,un      ;Сообщение о выгрузке
    int     21h
    retn
set_int    endp
msg    db    0dh,0ah,'Программа уже в памяти, /u -
выгрузка',0dh,0ah,'$'
msgno    db    0dh,0ah,'Программы нет в памяти',0dh,0ah,'$'
un    db    "Программа выгружена",10,13,'$'
Code     ENDS
END      resprog

```