

ZX-Spectrum-48K в эпоху windows и эмуляторов.

ЧАСТЬ 1

Настройка эмуляторов и обзор ПО под windows.

Предисловие.

Посвящается всем поклонникам Спектрума 48К, и их многочисленных клонов, до появления на них дисководов и TR-DOS.

Первым моим компьютером, был «Дубна 48К» с кассетным магнитофоном. Спустя пару лет, в 1994 году, у меня появился Спектрум 128 с дисководом. Но за это время я настолько привык к версии 48К, что быстрая загрузка с дисковода и стартовое меню TR-DOS, меня сильно разочаровали. С такими доработками домашний компьютер, больше походил на IBM-совместимый, и терял всю свою привлекательность. А через 3 года появился Pentium 166 с системой Windows 95.

Лично мне дорог как память, Спектрум с магнитофоном, и самые яркие воспоминания связаны именно с ним, и загрузкой программ с кассет. Со временем по «пищанию» начинаешь различать, что именно грузится на компьютер. По характерным звукам можно было определить прорисовку картинки, машинные программы, текстовую информацию, загрузку русских шрифтов, и так далее.

Да и само ожидание загрузки, стоило многого. Поставив на загрузку игру с переписанной у кого-нибудь, и затертой до дыр кассеты, с напряжением смотришь, загрузится она с первого раза или нет. За это время от волнения успеваешь выпить кружку чая. И вот, долгожданная игра запустилась, жмешь кнопку «СТОП» на магнитофоне, ставишь кружку на стол, и радостно протягиваешь руки к клавиатуре...

Настоящего Спектрума давно нет, все опыты и исследования проводились с эмуляторами, и разнообразными утилитами для работы под Windows XP и 7 в 2013 году. Изначально книжка задумывалась как современная пародия на старые самоучители: «Основы программирования на языке BASIC». Писал исключительно для себя, чтобы не забыть интересующие программы и функции, но по мере разрастания глав, решил аккуратно оформить, как книжку, для общего обозрения. Понимаю, что для большинства это давно не актуально, но если хоть несколько человек найдут для себя что-то интересное и полезное, будет прекрасно.

Глава 1.

Обзор преимуществ и недостатков эмуляторов.

Для проведения опытов, и создания псевдо-кассетных программ нам понадобятся несколько разных эмуляторов. Существует много разных эмуляторов. Но в каждом есть свои достоинства и недостатки.

Для воспроизведения, создания, и тестирования программ Спектрума под Windows, мне очень нравится Spectaculator v7.0.1.1349 (можно и более раннюю или позднюю версию), настроенный на режим считывания файлов кассетного магнитофона в реальном времени. Кроме того, эмулятор умеет записывать *.tap и *.tzh файлы. Этот эмулятор будет использоваться в качестве основной программы для работы с данными.

Интересным может показаться эмулятор Realspectrum v0.97.31. Он немного глючный, и давно не обновляется, но тоже есть удобная функция записи *.tap и *.tzh файлов. Этот эмулятор также умеет записывать файлы без заголовка с помощью процедуры из ПЗУ.

В EmulzWin версий 2.3 - 2.7 (Владимира Кладова) есть потрясающая функция ассемблера и дизассемблера. С ее помощью можно написать программу на ассемблере и

тут-же скомпилировать и протранслировать в память. Можно написать программу даже в блокноте windows и просто по *CTRL+C* внести в окно ассемблера. При этом не надо никаких вспомогательных программ, кучи лишних операций и компиляторов. Также из любой программы можно дизассемблировать любой кусок памяти, прямо в процессе работы программы, в виде команд ассемблера, или просто данных. Текст из окна ассемблера можно скопировать в буфер обмена, перенести и сохранить в обычном блокноте. Понятно, что таким образом можно изучить и подредактировать уровни любимой игры, в реальном времени на эмуляторе, не прибегая к редакторам и специальным программам.

Ассемблер в этом эмуляторе немного глючный, и в случае малейшей ошибки не хочет транслировать программу, не показывая сообщения об ошибках. Есть и некоторые другие недочеты. К сожалению, в следующих версиях, вместо доработки, эта удобная функция была зачем-то убрана. *В Windows 7 при запуске эмулятора 2.7 возможны проблемы, поэтому запускать программу нужно в режиме эмуляции XP. Также замечен конфликт с Internet Explorer при работе эмулятора.*

Эмулятор ZX-Spin v0.7 тоже имеет ряд преимуществ. У него тоже имеется встроенный ассемблер, но менее удобный, чем-то похожий на стандартные ассемблеры. Он также умеет записывать видео и аудио.

Глава 2.

Настройка Spectaculator для загрузки с магнитной ленты в реальном времени.

Чтобы эмулятор загружал *.tap и *.tzh файлы в реальном времени, а не мгновенно, после установки, необходимо покопаться в его настройках. По умолчанию эти опции отключены, и многие даже не подозревают о них.

Запустим эмулятор и произведем предварительную настройку. Первым делом нажмем *CTRL+K* (или кнопку с кассеткой на панели). Рядом откроется окошко «*Cassette Recorder*». Это виртуальный магнитофон. Расположите его рядом с окном эмулятора, как вам удобно. Перетаскивая в него мышкой файл *.tap или *.tzh, вы заряжаете в него кассету с программой. Кнопочки на панели магнитофона говорят сами за себя.

Далее сделаем рамку на экране приемлемого размера. Для этого нажмем «*View*», в открывшемся меню «*Border*» и выставим черную точку на «*Medium*»:

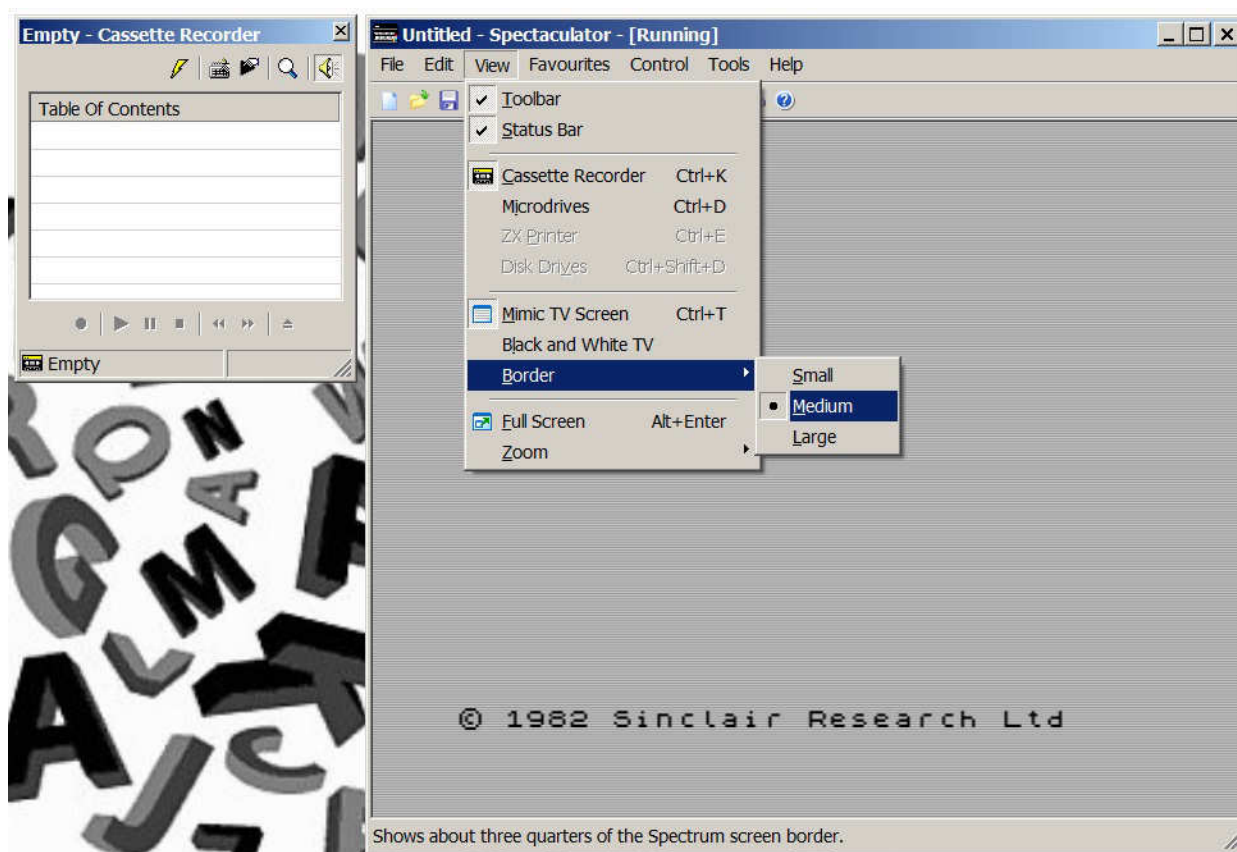


Рис. 1200. Окно виртуального кассетного магнитофона и настройка рамки в эмуляторе Spectaculator.

Теперь опции считывания магнитной ленты. Войдите в меню «*TOOLS*» и выберите «*OPTIONS*» или нажмите *CTRL+Y* в эмуляторе. Откроется окно с настройками. Нажмите на вкладку «*Cassette Recorder*» и первым делом снимите галочку «*Enable Fast Loading*», и двух нижних опций, которые с ней связаны:

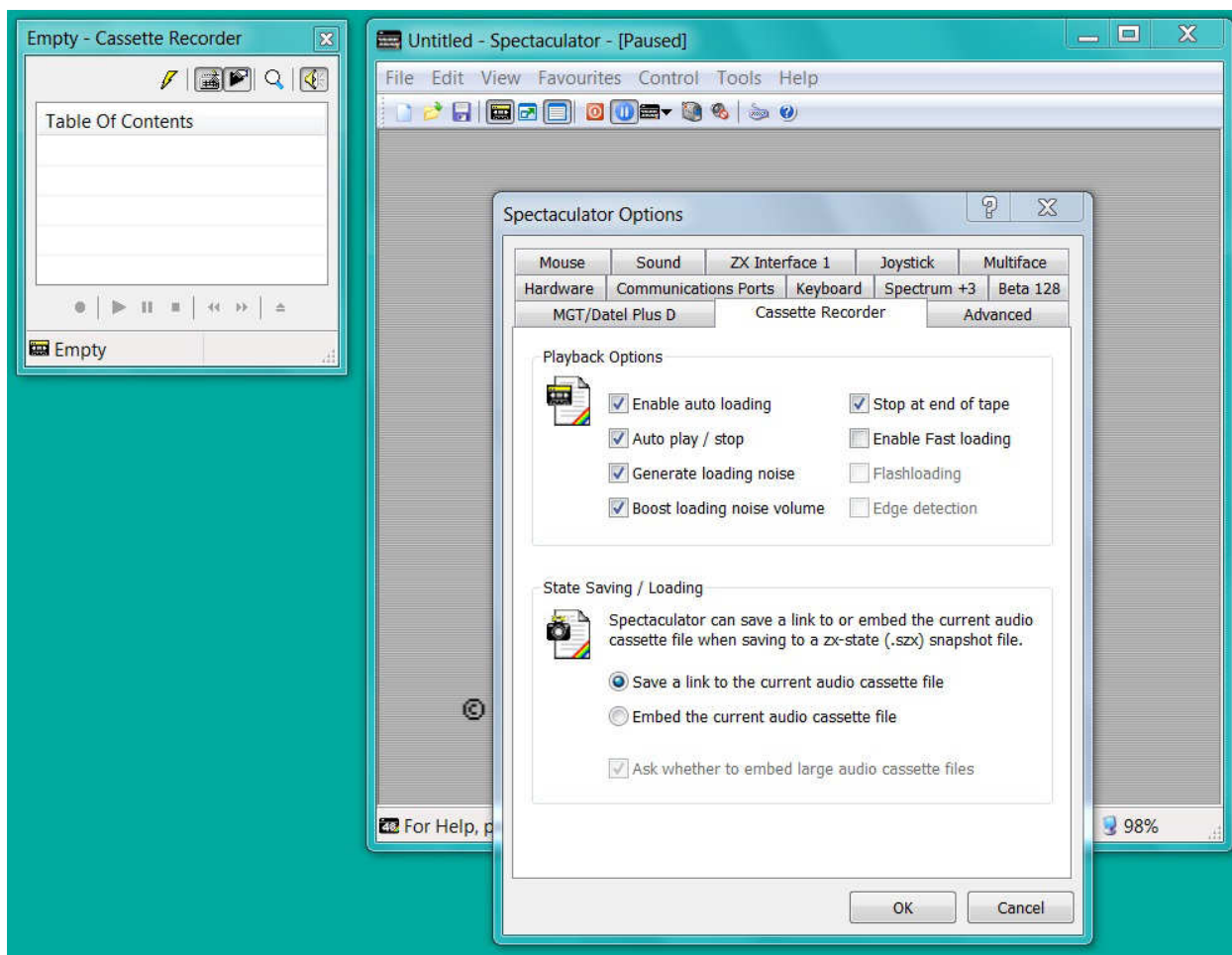


Рис. 1201. Настройка опций воспроизведения с магнитной ленты в реальном времени.

В нашем случае, самые главные опции это «*Generate loading noise*» и «*Boost loading noise volume*». Остальные галочки можно настраивать по собственному вкусу. Если же нравится полностью неавтоматический режим, когда приходится включать/выключать магнитофон кнопкой, и набирать LOAD "", то можно оставить настройки, как показаны на рисунке. Для подтверждения настроек нажмите «OK».

Установки по части магнитофона завершены. Некоторые опции можно менять в виртуальном кассетном магнитофоне. Например, если приходится несколько раз проверять длинную программу, а загружать в реальном времени надоело, можно во время загрузки нажать кнопку с пиктограммой «Молния», расположенную по центру и программа моментально догрузится, с ближайших битов сигнала данных. Только после загрузки кнопку необходимо отжать.

Часть 2.

Создание *.tap и *.tzh. файлов эмуляторами на PC под windows.

Глава 1.

Создание самозапускных блоков «Bytes:» с BASIC программой внутри.

Краткое содержание:

основы работы в эмуляторе Spectaculator, самозапускной блок с BASIC программой, работа в отладчике «Debugger», слепок памяти *.z80, системные переменные BASIC, создание *.tap файла, работа с виртуальным магнитофоном.

В некоторых игровых программах, например, BLIND ALLEY, загрузка начинается с `LOAD ""CODE`, без блока «Program:». Программа начинается с картинки, переходит в блок кодов, никаких `RANDOMIZE USR ...` Во время загрузки программа легко вскрывается, но никакого BASIC'а там не обнаруживается. Мало того, она как-то сама запускается после загрузки без дополнительных строк BASIC. А ведь для блока «Bytes:» никаких команд автостарта не предусмотрено, как например, для блока «Program:», где при записи следом можно поставить оператор `LINE` с номером строки. После `CODE` это сделать невозможно. Давайте не только разберемся, как это делается, а разовьем идею дальше. Помимо автостарта в загружаемый блок «Bytes:» поместим обычную BASIC программу.

Откроем Spectaculator, и создадим простейшую программу на Basic со строками. Например, такую:

```
1 PRINT INK 1;"TEST"  
2 BORDER 6  
3 PAUSE 0  
4 BEEP 1,0
```

На экране компьютера эта программа будет выглядеть следующим образом:

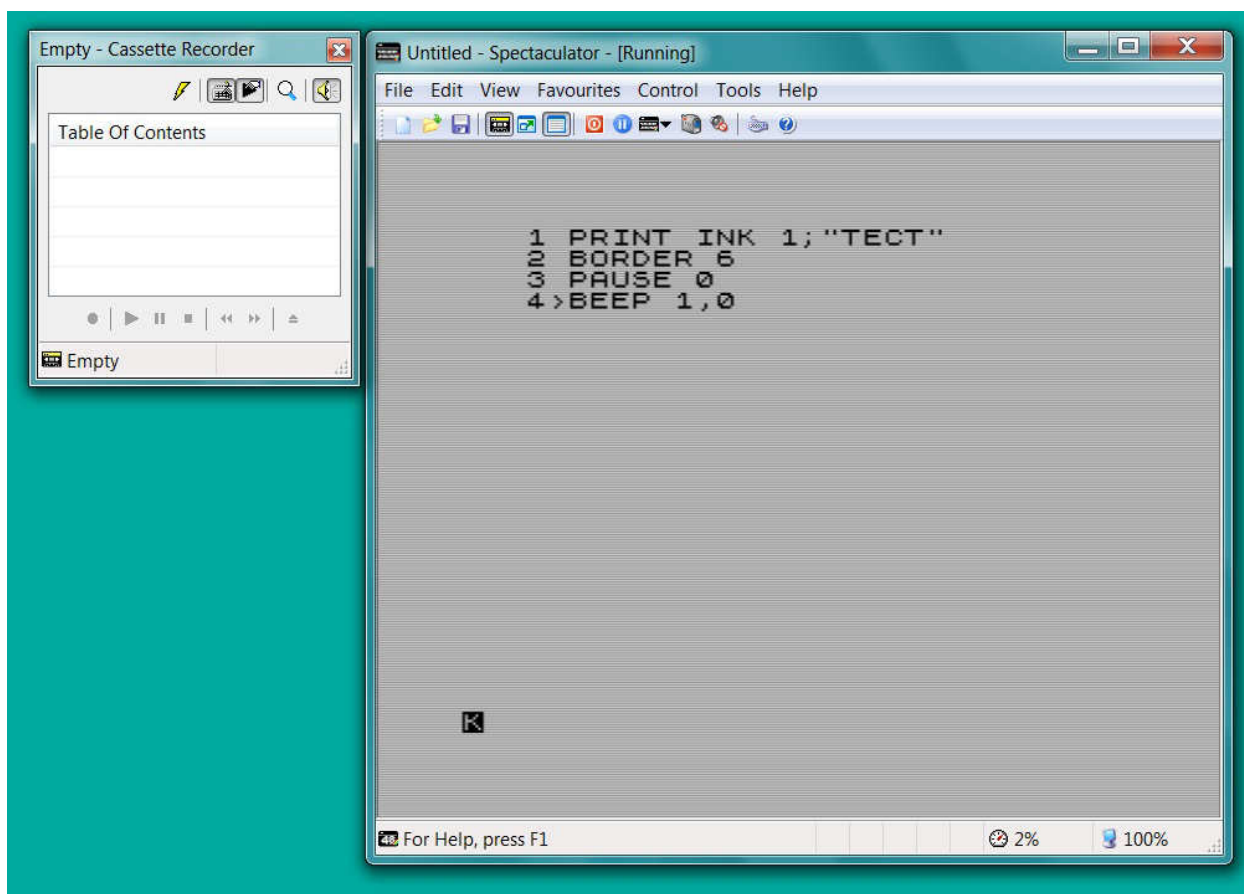


Рис. 2100. Тестовая BASIC программа на экране эмулятора Spectaculator.

По желанию, можно проверить ее работоспособность. Теперь в нижней строке редактора нужно написать подготовительную программу, которая должна выглядеть так:

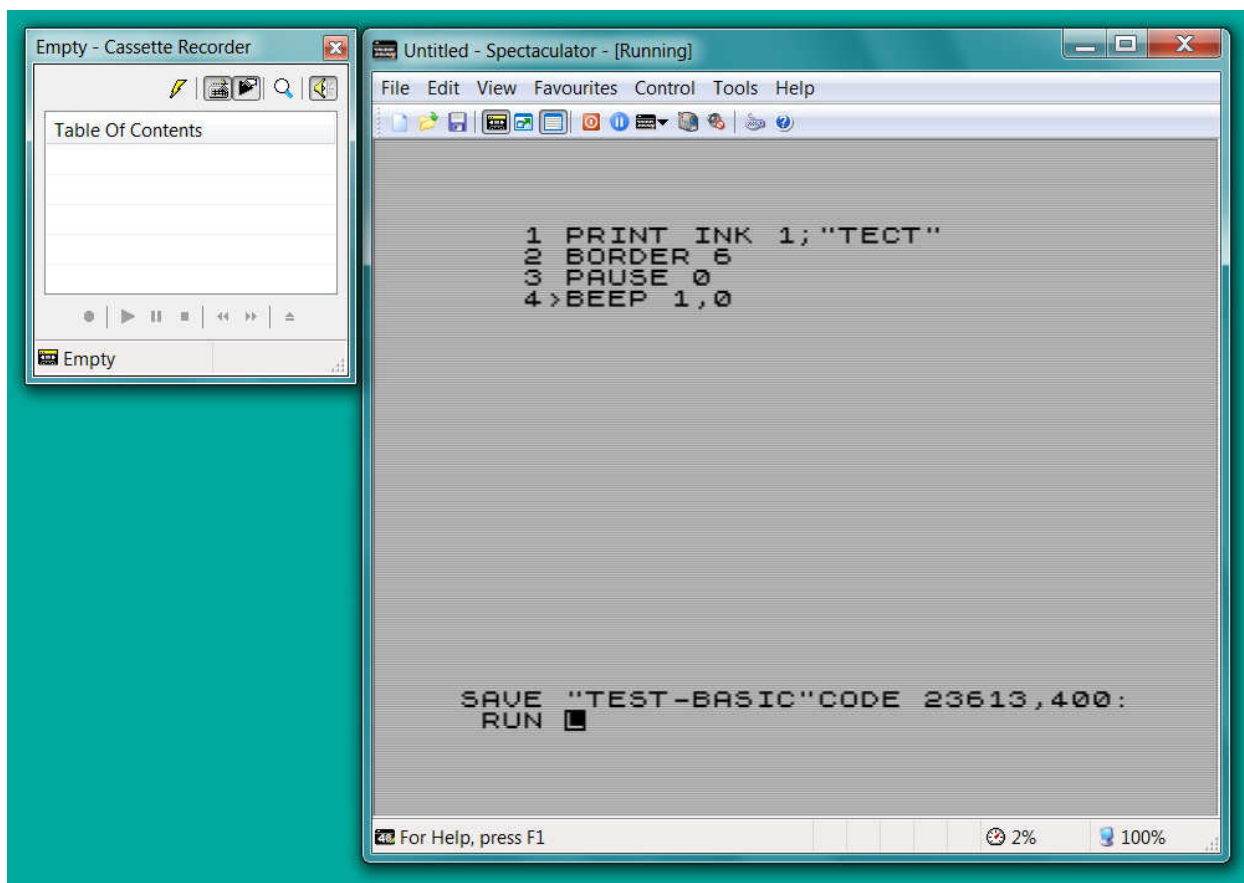


Рис. 2101. Spectaculator: Подготовительная строка с примерным размером данных.

Длина в 400 байт взята пока на глаз, чтобы в дальнейшем подкорректировать. Главное чтобы при этом не изменилась длина строки. Если программа большая, то для резервирования значения длины лучше вписать 4 цифры, например 1000.

Теперь подробнее о стартовом адресе. С адреса 23552 размещена область системных переменных BASIC. Таблица и подробное описание есть в приложениях почти каждой книжки по BASIC'у или Ассемблеру, лежащей в интернете (*например, обширная электронная библиотека русскоязычных книг по ZX-Spectrum есть на <http://vtrdos.ru>.*). Адрес 23613 это максимальный, ниже которого появляются критические переменные, изменяющиеся в процессе работы операторов в программе. Значения некоторых из них отличны от тех, которые записываются при старте компьютера. Поэтому для корректной работы простеньких программ лучше записывать BASIC вместе с этими переменными.

А теперь самое главное длина будущей программы, и как ее вычислить. В Spectaculator нажимаем **CTRL+ENTER**, или открываем вкладку «Tools», а в выпадающем меню выбираем «Debugger». На мониторе появятся еще два окошка. Которое поменьше, это «Registers», с текущими значениями регистра процессора Z80, большое – окно отладчика «Debugger». Расположите их как удобно на своем рабочем столе.

В первый раз окна откроются с шестнадцатичными значениями адресов и их данных. Переключение в десятичный режим осуществляется выключением кнопочки «HEX» в окне отладчика, или нажатием **CTRL+H**. На экране должно появиться примерно следующее:

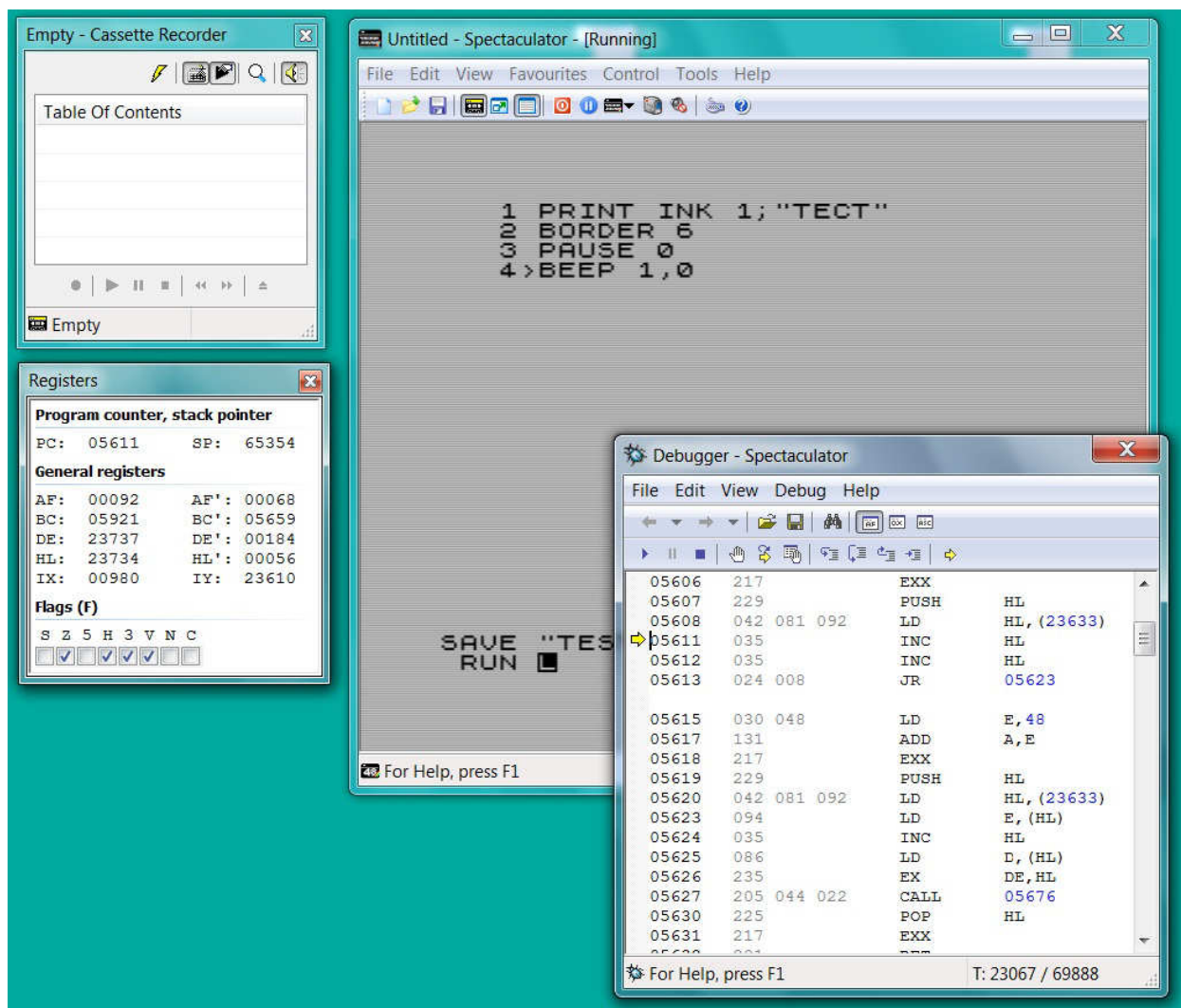


Рис. 2102. Окна отладчика (Debugger) с курсором, и значений регистров (Registers) эмулятора Spectaculator

Желтый курсор отладчика указывает текущую выполняемую машинную команду. Теперь задача отыскать адрес размещения BASIC строки, ожидающей ввода. Она находится позади всех строк программы, а следовательно она будет конечной. Так как размер программы, величина не постоянная, в области системных переменных бейсика существует двухбайтовая переменная **E_LINE**, которая находится по адресу 23641 и 23642. Она укажет адрес начала строки в нижней части экрана, ожидающей ввода. В данном случае такой строкой является:

```
SAVE "TEST-BASIC" CODE 23613,400: RUN
```

Хватаем курсором мыши за «лифт» и прокручиваем в текст в отладчике до нужного адреса, не трогая желтый курсор, устанавливаем мигающую палочку, и видим значения нужных ячеек 16 и 93. Вычисляем адрес строки путем умножения второго байта на 256, и прибавлением первого к полученному результату. В итоге должно получиться 23824.

Теперь обратимся к переменной **WORKSP**, которая расположена по адресу 23649 и 23650. Она указывает на следующий «слой» вспомогательных данных бейсик программы. Там располагается рабочая область операционной системы. Она укажет нам на начало следующих за ожидаемой ввода строкой, данных. В настоящий момент мы увидим там байты 43 и 93. Эти числа указывают на адрес 23851.

Переходим к делу. Находим в *Debugger*'е адрес 23824. И действительно, самым первым по адресу 23824 стоит число 248, которое соответствует шифру команды `SAVE` из таблицы команд:

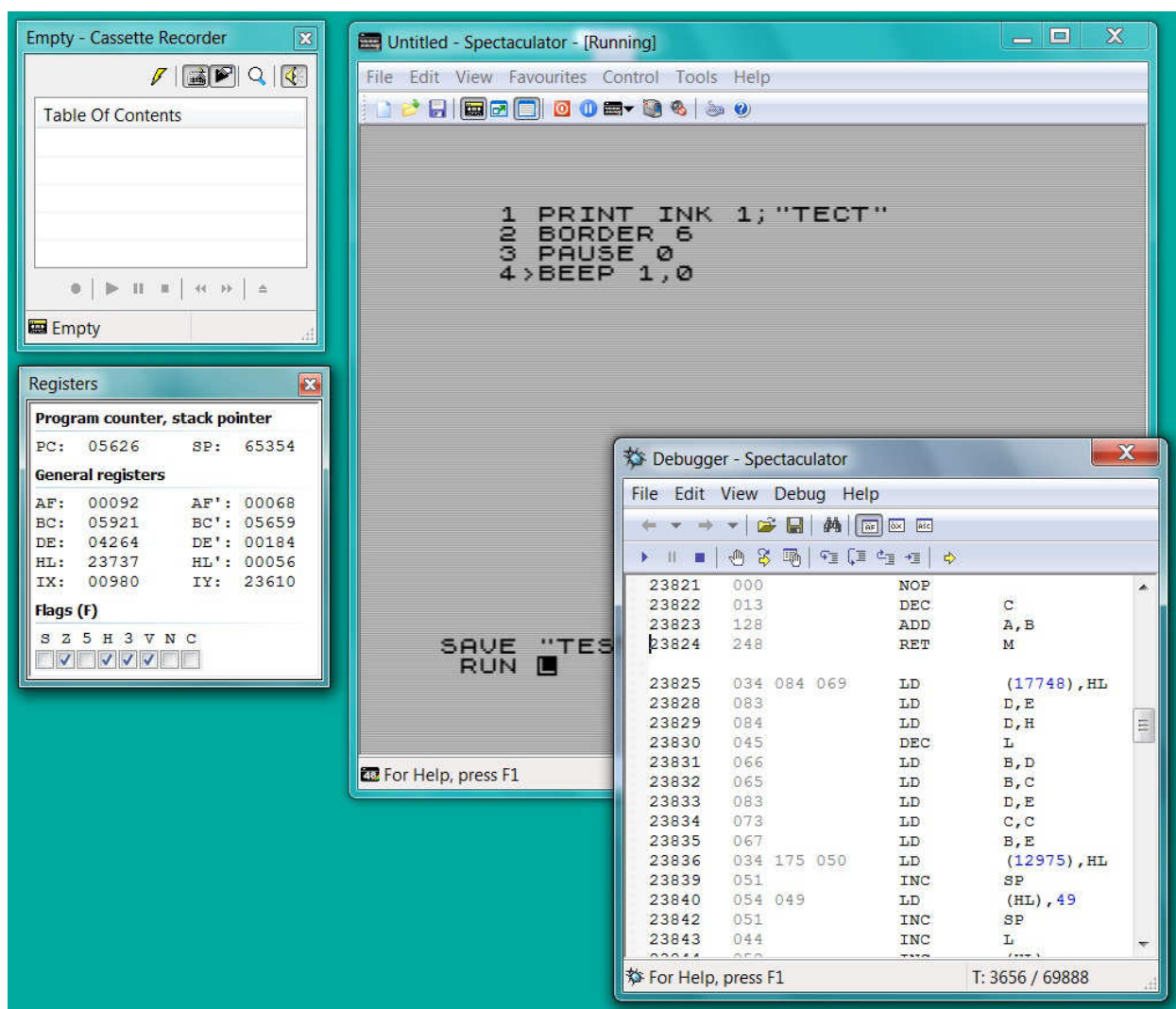


Рис. 2103. Окно отладчика *Debugger*. Поиск нужной строки. Тонкий курсор.

Окончание программы тоже теоретически вычислили. Проверим на практике, так ли это. Внимательно осматриваем каждый байт, выясняя, где заканчивается эта строка. Для удобства поиска, с помощью `CTRL+A` можно включать/выключать режим просмотра символов в ASCII (или самая правая кнопочка `ASC` в 1-м ряду инструментов). В этом режиме в столбец отобразится имя заголовка вводимой строки «TEST-BASIC» и числа после команды `CODE`. Выйдя из режима ASCII (если заходили) по адресу 23248 обнаруживаем 247 (числовой эквивалент команды `RUN`), следом 13 (`ENTER`), а затем маркер «128». Предварительный адрес конца программы полностью совпадает с предположенным. Им является ячейка 23850:

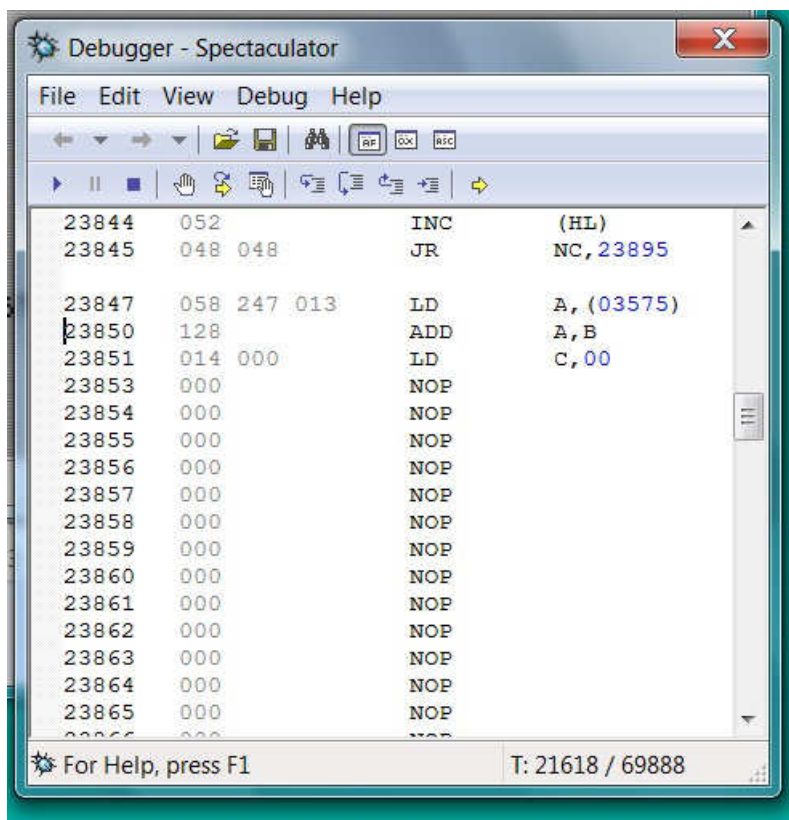


Рис. 2104. Окно отладчика Debugger. Окончание BASIC строки в нижней части экрана.

Следует помнить, что после редактирования каждой строки, эта область не всегда обнуляется. Перекомпоновка происходит только во время запуска BASIC программы или ее увеличения. Часто строка затирается по мере надобности. Если до этого момента редактировалась строка с большей длиной, то хвост ее останется незатертым текущей строкой.

Также стоит обращать внимание на переменные **STKBOT** (23651 / 23652) – стек калькулятора и **STKEND** (23653 / 23654) – стек операционной системы. В сложных BASIC программах с вычислениями, циклами, массивами и подпрограммами, эти области также могут сильно «распухать». Но в данном случае, с простейшей программой, все три переменные **WORKSP**, **STKBOT** и **STKEND**, не используются, и поэтому, указывают на один и тот же адрес – 23851. В данном опыте они не понадобятся, но в будущем они могут очень пригодиться. Вычисляем черновую длину $23850 - 23613 = 363$ байта.

Но это еще не все, следует учесть, что после нажатия **ENTER** и ввода строки, пойдут процессы, которые отследить до записи не удастся. Программа расползется на какое-то количество байт, потому что выполнятся операторы в этой строке, например тот же **RUN**. При этом расширяется область переменных **VARs**, которая вклинивается в область между программой на бейсике и вводимой строкой. Поэтому нужно взять длину с небольшим запасом. Дополнительная длина может быть разной, в зависимости от стоящих операторов, через двоеточие после **SAVE**. Чем их больше, тем дальше сползет конец программы вниз.

Если в строке после **SAVE "TEST-BASIC" CODE 23613, 400**, через двоеточие стоит только **RUN**, то «индекс сползания» будет 12 байт, а если длина иная, то нужно вычислить индекс, и прибавить к его основной длине.

Давайте разберемся на простом примере, как это происходит. Сохраните программу текущим слепком памяти, в формате *.z80, например, под именем «test-basic.z80». Для этого в эмуляторе откройте вкладку «File → Save as...» В строке «Save as type» выберите «Z80 Snapshot (.z80)»:

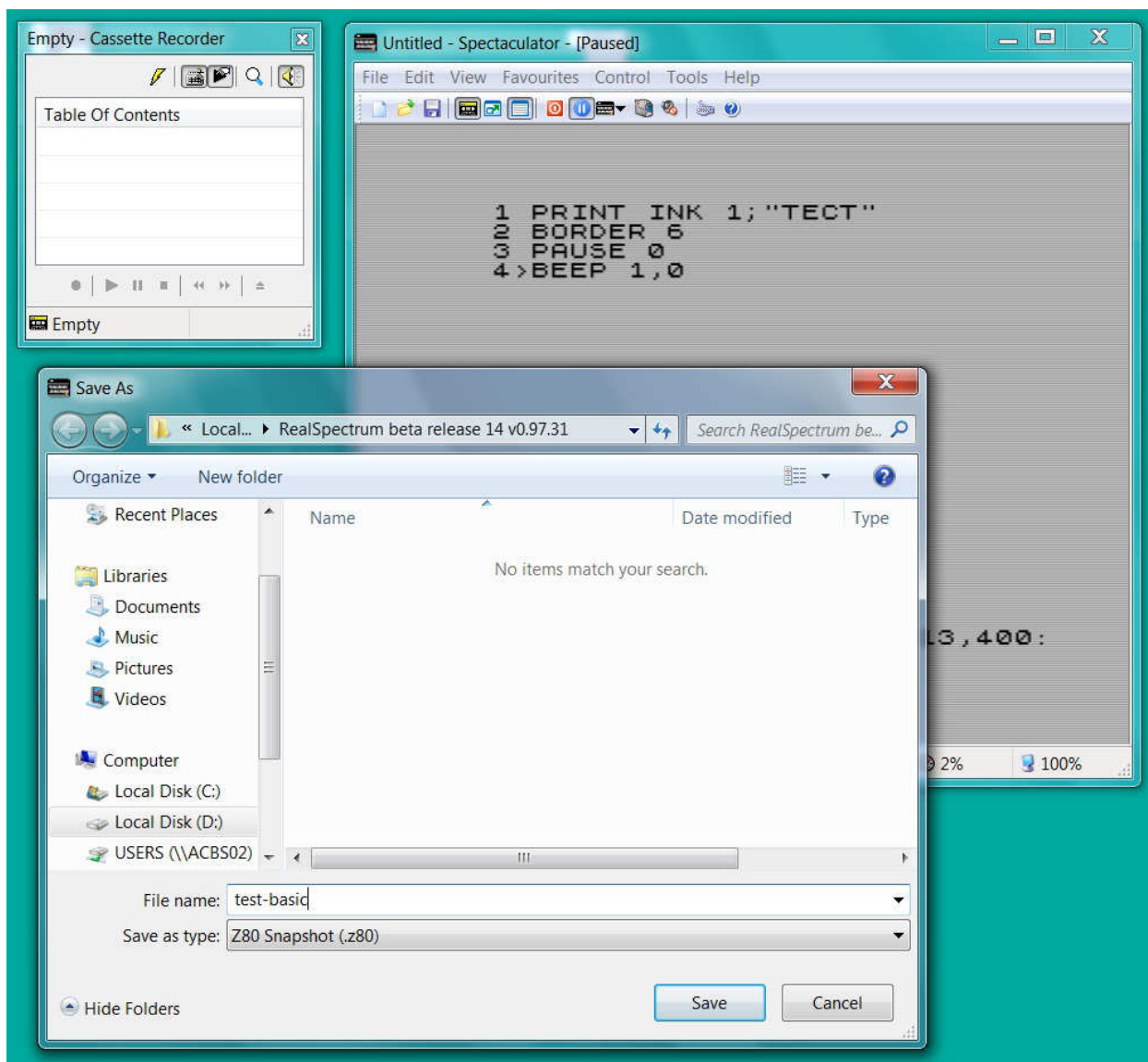


Рис. 2105. Сохранение слепка памяти с текущими значениями.

Введите строку своей программы на Spectrum. Появится надпись «start tape, then press any key». Снова нажмите **ENTER**, и терпеливо подождите, пока выполнится вся загрузка. Затем еще раз нажмите любую клавишу для **PAUSE 0**, и на фоне желтой рамки выскочит сообщение **OK, 4:1**.

Теперь после прогона «сценария развития событий» сразу откройте *Debugger*, и посмотрите новое значение длины.

Обратите внимание, что переменная **WORKSP** теперь уже нам в этом не помощник. После выполнения строки в нижней части экрана, компьютер про нее забыл и передвинул все сзади стоящие области на пару десятков байт выше. Строка стала отработанным мусором, и поэтому при следующей операции начнет затираться, по мере надобности. В этом случае следует искать адрес только вручную. Всегда стоит внимательным образом искать конец действующей строки, и не перепутать с остаточным мусором от предыдущих операций (если таковые были).

Как видите, данные после выполнения сползли на адрес 23862, то есть на 12 байт ниже. Вот это и есть тот самый «индекс сползания программы».

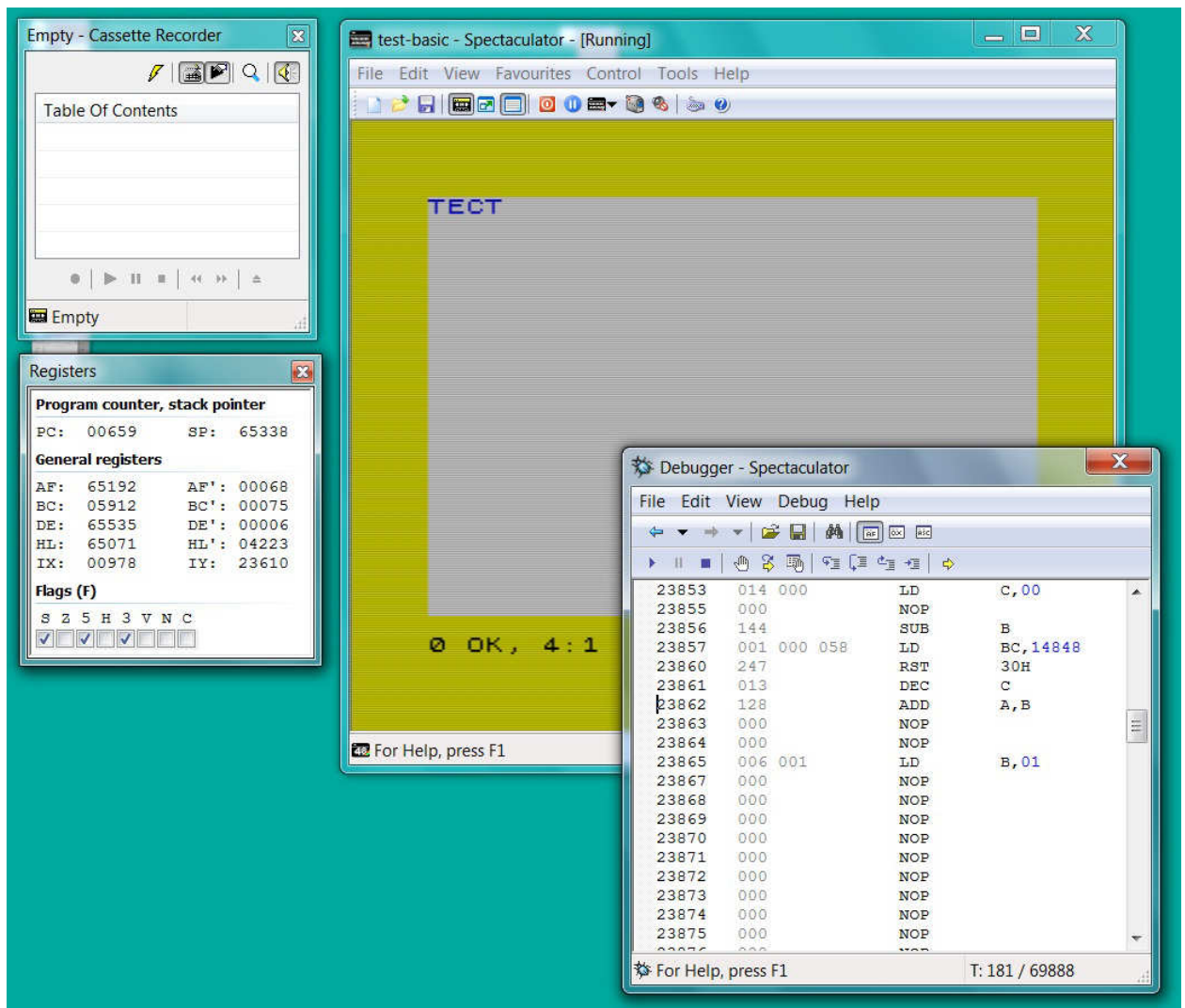


Рис. 2106. Окно Debugger. Смещение адреса конца программы после тестового прогона.

Выставив окончательное значение, после корректировки +12 получаем гарантированную длину программы, с учетом смещения, после выполнения будущей строки. Окончательная длина станет 249 байт. Если терзают сомнения, можно прибавить пару байт для надежности. Если вы недогрузите программу, хоть на байт, автозапуска может не произойти. После загрузки, в таком случае может выскочить сообщение: `␣ Nonsense in BASIC ␣: x`, где `x` - номер оператора.

Теперь возвращаемся к программе, и загружаем в эмулятор слепок памяти «*test-basic.z80*». Для этого в эмуляторе последовательно откройте вкладки «*File* → *Open*», и выбираем сохраненный файл «*test-basic.z80*». Мгновенно программа, со всеми значениями регистров, и с того самого места, на котором была сохранена, вернется на Spectrum, как будто ничего и не произошло. В нижней строке мы увидим строку `SAVE...` (таким же образом можно проходить сложные игры на спектруме, периодически сохраняя слепки памяти, перед прохождением сложных моментов, а в случае неудачи откатывать назад). Курсор в нижней строке мигает, ожидая ввода. Выставляем окончательную длину, с учетом смещения, заменив 400 на 249, и переведем курсор в конец строки. Она должна выглядеть следующим образом:

```
SAVE "TEST-BASIC"CODE 23613,249: RUN
```

Сохраняем окончательную заготовку под именем «*test-basic.z80*», перезаписав предыдущую версию. Перед нами «полуфабрикат», готовый для преобразования в «*Bytes* : », формата **.tap*, **.tzx*, **.wav* или **.csw*.

Теперь в окне эмулятора нажимаем кнопочку с видеокамерой, или комбинацию **ALT+F8** на клавиатуре. Откроется окно «New», в котором отобразятся 10 листочков с шаблонами, для записи будущего файла. Наведем мышкой на предпоследний «Audio cassette file (.tap)», и выберем его. Справа отобразится описание создаваемого файла. Вверху, в окошке «File», будет стоять имя файла с выбранным расширением. Если вы до этого ничего не записывали, то по умолчанию там увидим «Untitled1.tap». Переименуйте файл в «test-basic1.tap». При желании в окошке «Location» можно выбрать путь, куда сохранять созданный файл:

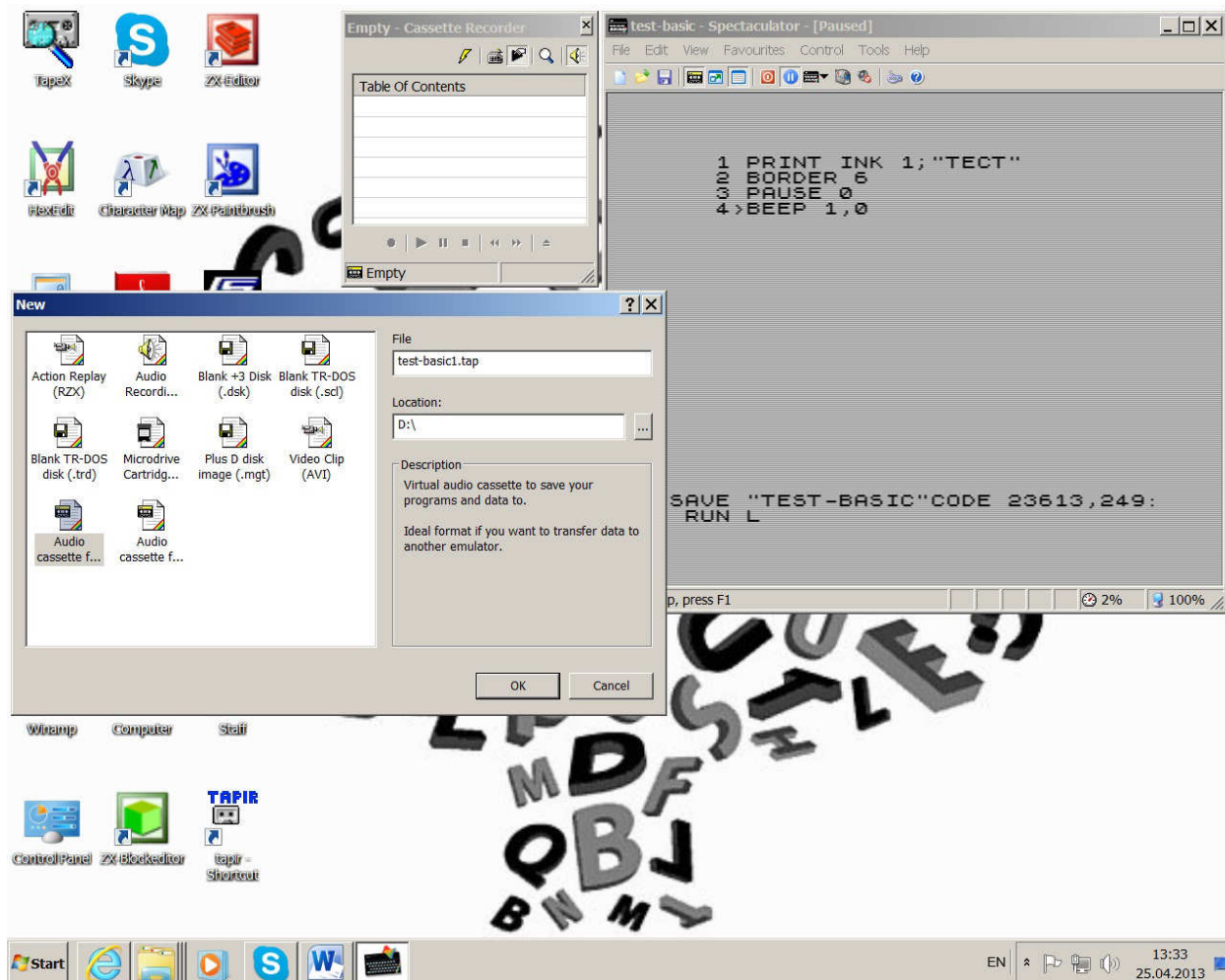


Рис. 2107. Окно «New». Выбор параметров записи и имени будущего файла.

Нажмите «OK» и окно исчезнет. Теперь в визуальном магнитофоне нажмите кнопочку с красной точкой «Record». Во 2-м квадратном окошке эмулятора появится значок кассеты с красной точкой. Это означает, что эмулятор готов к записи:

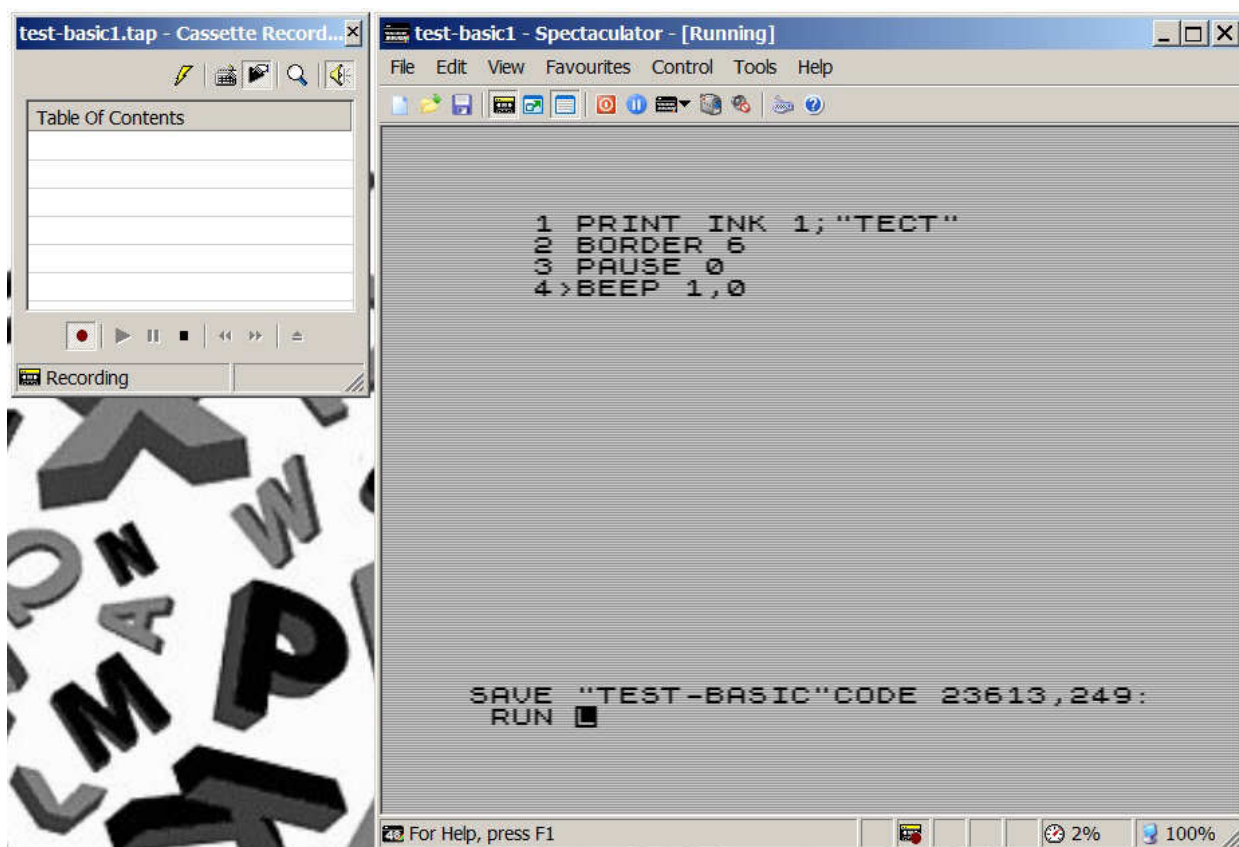


Рис. 2108. Готовность эмулятора к работе. Значок кассеты с красной точкой.

В эмуляторе введите свою строку, нажав **ENTER**. В нижней части экрана появится надпись: «Start tape, then press any key»:

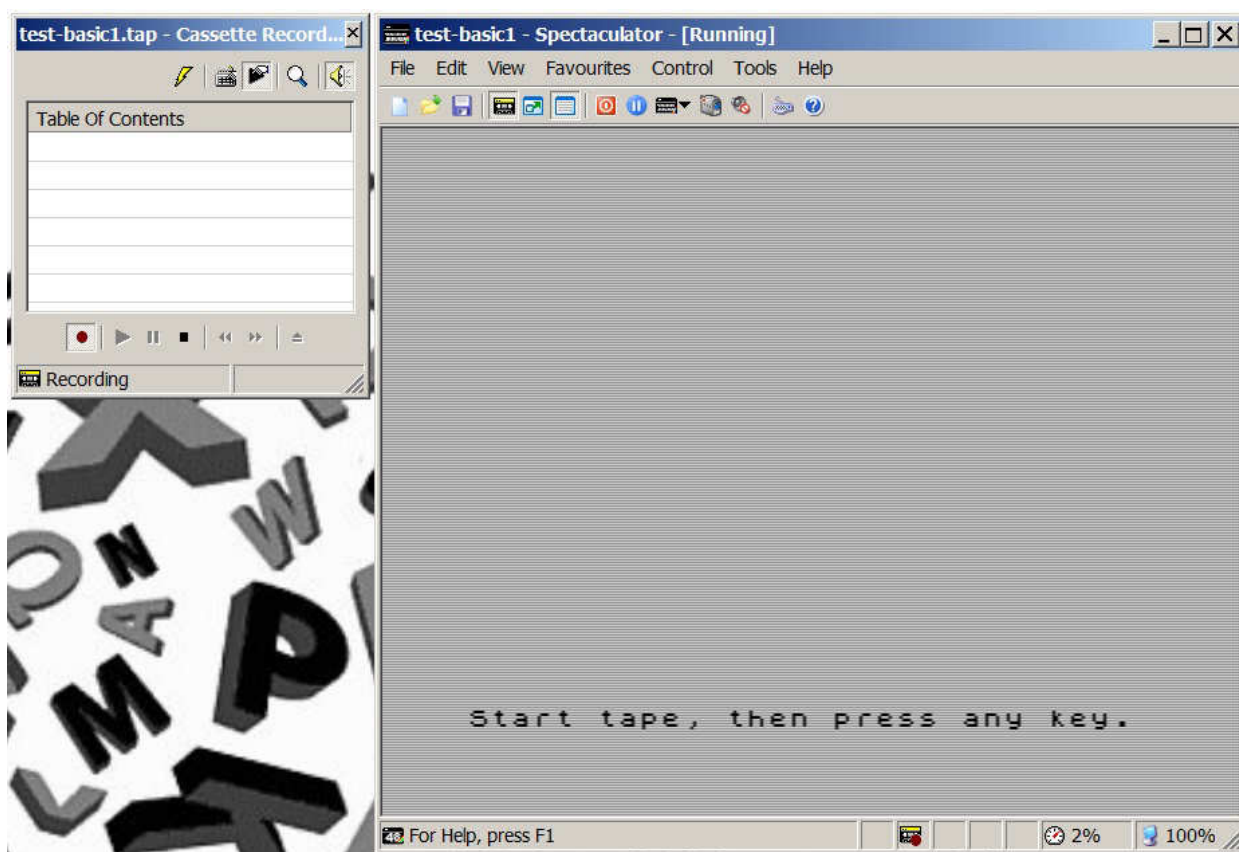


Рис. 2109. Готовность программы к записи. «Start tape, then press any key.»

Нажмите еще раз **ENTER**. В окне виртуального магнитофона появится заголовок с блоком данных, а на экране эмулятора пойдет дальнейшее выполнение программы. В данном случае, выскочит желтая рамка, и компьютер будет ожидать нажатия любой клавиши:

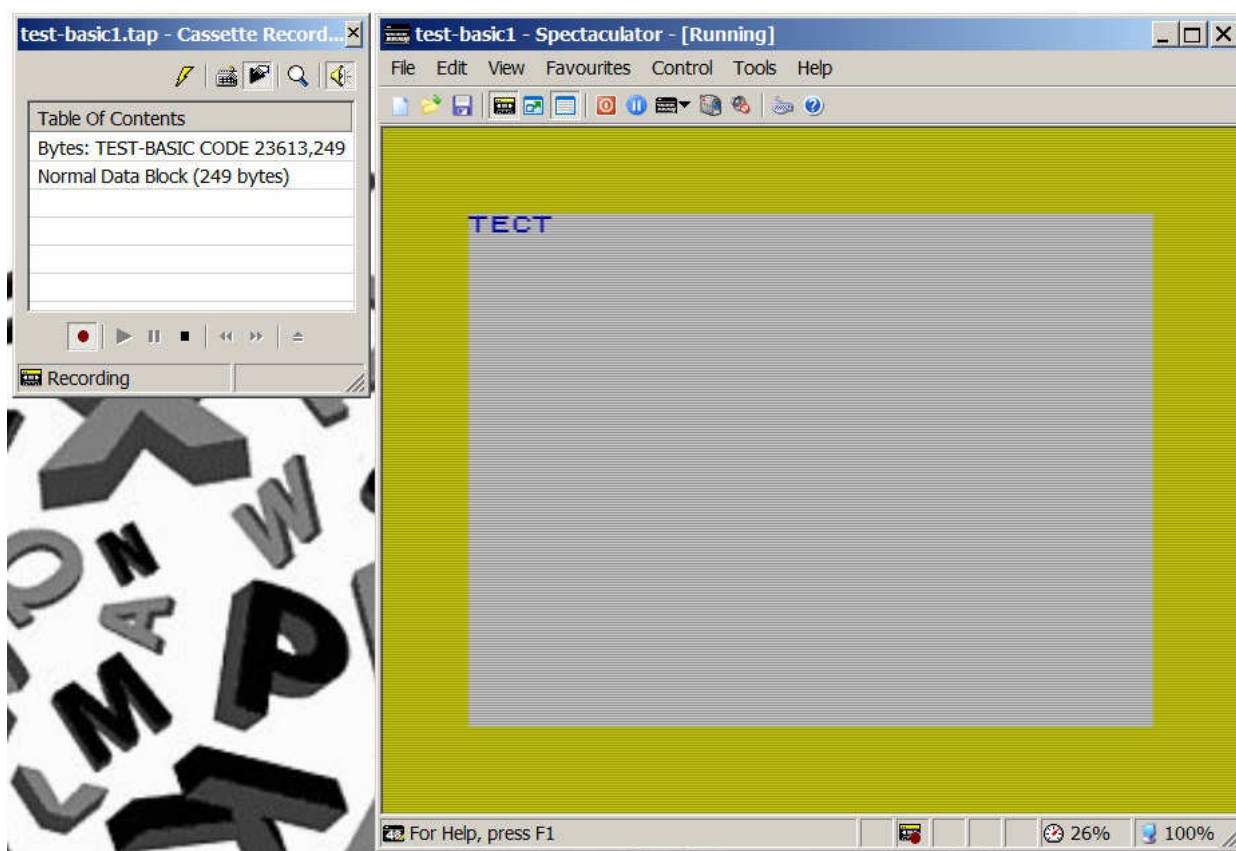


Рис. 2110. Продолжение выполнения программы после успешной записи.

Нажмите «Stop» на виртуальном магнитофоне, и файл готов. В нижнем маленьком окошечке кассета с записью исчезнет. Нажав любую клавишу в окне эмулятора, выдастся сообщение «OK, 4 : 1». Нажмите *Reset* на эмуляторе. Записанная программа, в виде двух строчек, останется в магнитофоне:

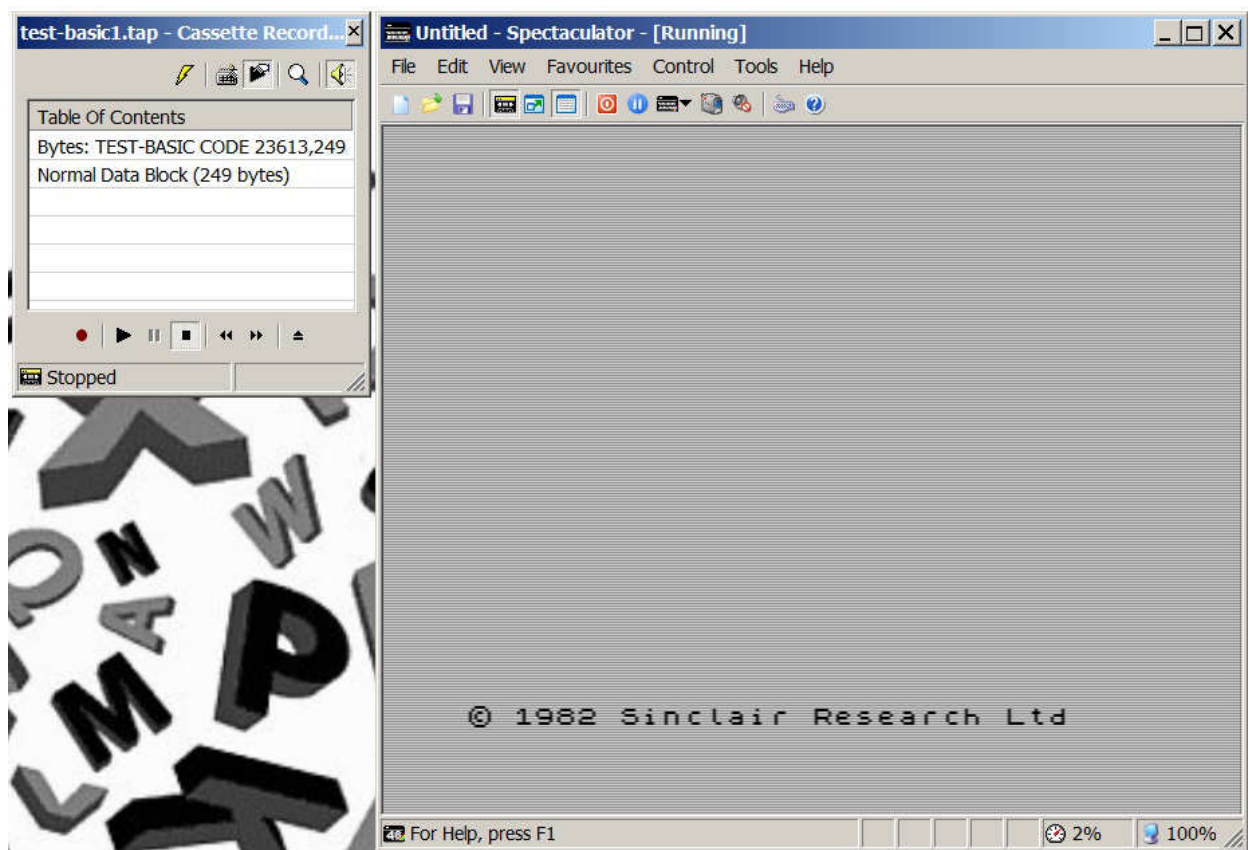


Рис. 2111. Spectaculator. Сохраненная программа в виртуальном магнитофоне.

Теперь будем проверять готовую программу. Для загрузки файла, в окне эмулятора введите строку `LOAD ""CODE` и `ENTER`. Мышью включите «PLAY» виртуального магнитофона. Из колонок польется многим знакомый звук загрузки файла. Только в отличие от заезженных кассет, он будет чистым и звонким:

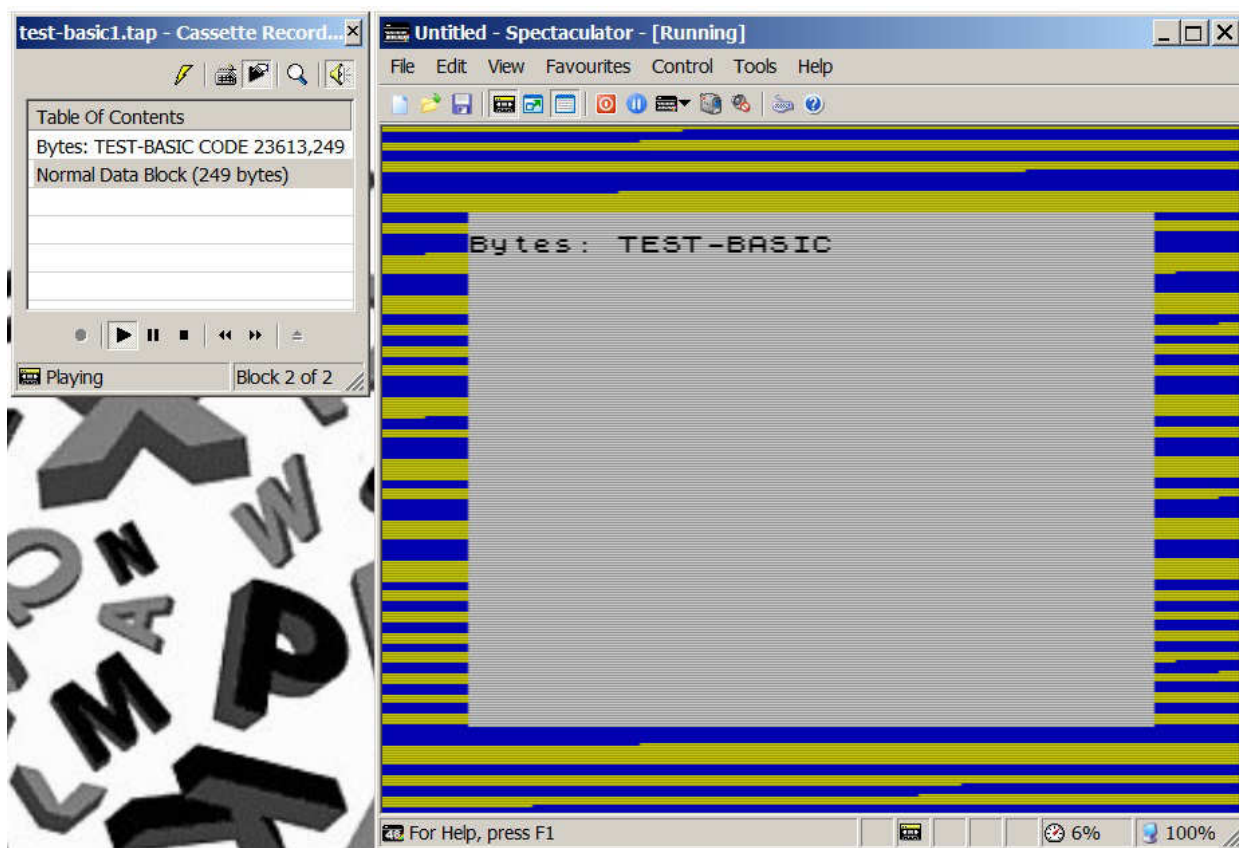


Рис. 2112. Проверка загрузки и работоспособности программы, созданной в виртуальном магнитофоне.

После загрузки блока «**Bytes :**», когда с рамки эмулятора исчезнут желто-синие полосы, сначала выполнится вводимая строка в нижней части экрана, с оператором **RUN**, который в свою очередь автоматически запустит строки программы на **BASIC**.

Для удаления программы из магнитофона, после загрузки можно нажать кнопку «**Eject**», расположенную в левом нижнем углу (треугольник с подчеркиванием). Также старая программа автоматически удаляется, при загрузке в магнитофон новой.

Глава 2.

Создание *.tzh файла с помощью эмулятора Realspectrum.

Краткое содержание: программа с невидимыми строками, создание *.tzh файла в Realspectrum.

Теперь рассмотрим создание и запуск программ с «заглушкой», за которой стоят выполняемые строки на **BASIC**.

Например, загружается блок «**Program:**». После загрузки выполняется какая-то программа, и после окончания выдается сообщение «**OK**». После нажатия **ENTER**, в ней обнаруживается всего одна строка **1 POKE 23780,0**. Нажав **RUN**, программа снова выполняется, но по окончании, опять остается одна строка. Давайте попробуем создать такую программу.

Откройте Spectaculator, и наберите следующую программу:

```

1 POKE 16384,0
2 PRINT "Проверка работы скрытых строк
программы"
3 POKE 16384,255
  
```

Теперь подберем адрес, с которой начинается номер строки 2, и заменим 16384 на него. Откроем *Debugger* и перейдем к просмотру области памяти, начиная с адреса 23755. Для этого в окне отладчика просто нажмите *CTRL+G*. Откроется маленькое окошко «*GO TO*». В нем введите 23755 и нажмите «*ENTER*».

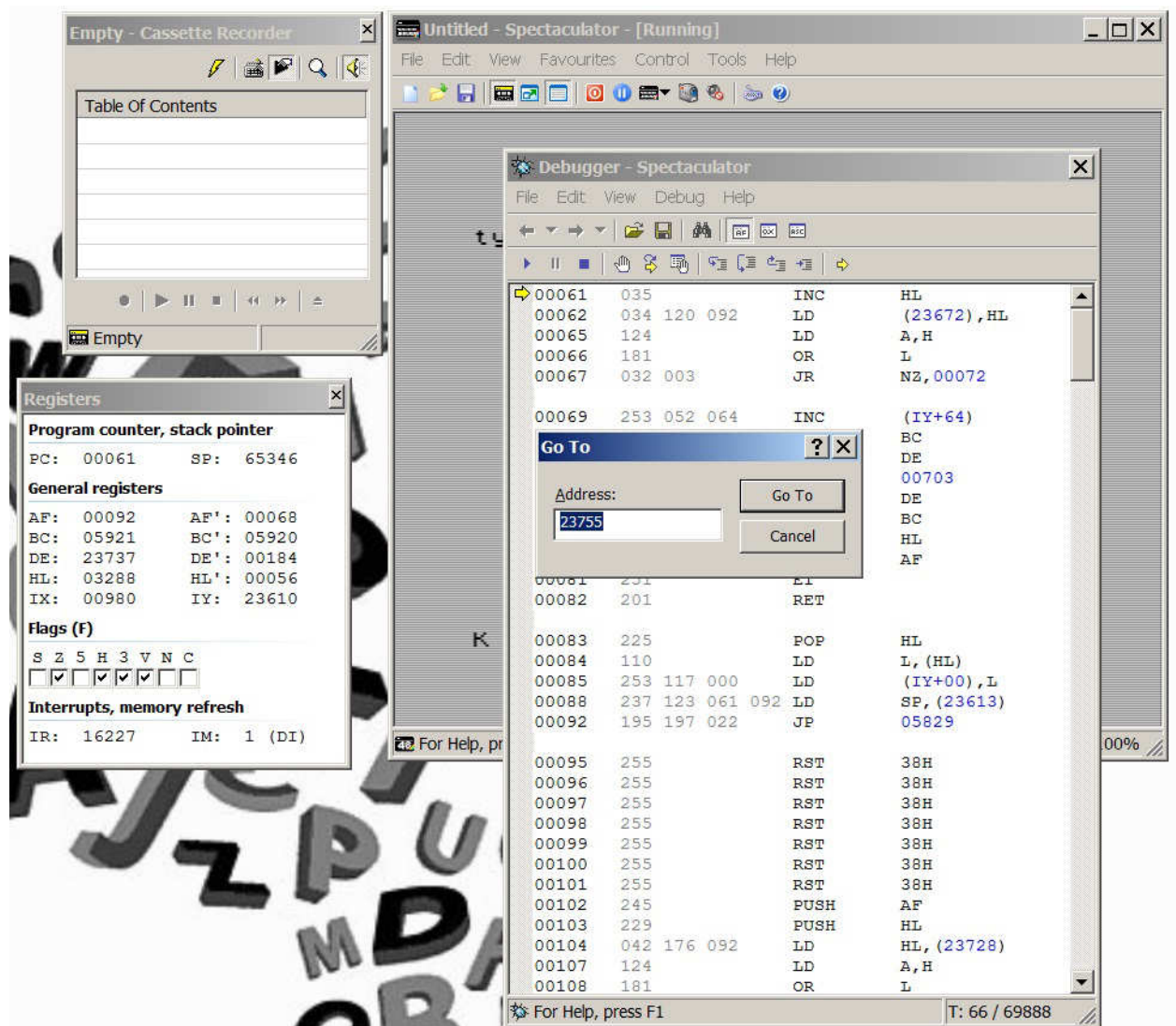


Рис. 2200. Поиск строк программы на BASIC. Быстрый переход к просмотру памяти по «*GO TO*».

Отладчик сразу откроет область памяти, начиная с адреса «23755». Осмотрев внимательно содержимое, находим нужную ячейку. Она будет сразу за числом «13» самой первой строки. Это адрес «23780». В итоге, в строках 1 и 3 подправим значения, и получим такую программу:

```
1 POKE 23780,0
2 PRINT "Proverka raboty skrytyh strok
programmy"
3 POKE 23780,255
```

Запустите программу и посмотрите на результат. Строки 2 и 3, исчезнут с экрана, но останутся в памяти. После снятия «заглушки» строки снова займут свое место, так как младший байт строки не повреждается.

Снова командой *RUN*, запусив программу. Программа выполняется, но строк не видно. Теперь запишем программу с функцией автостарта. Для этого наберите подготовительную строку:

SAVE "HID-BASIC" LINE 1

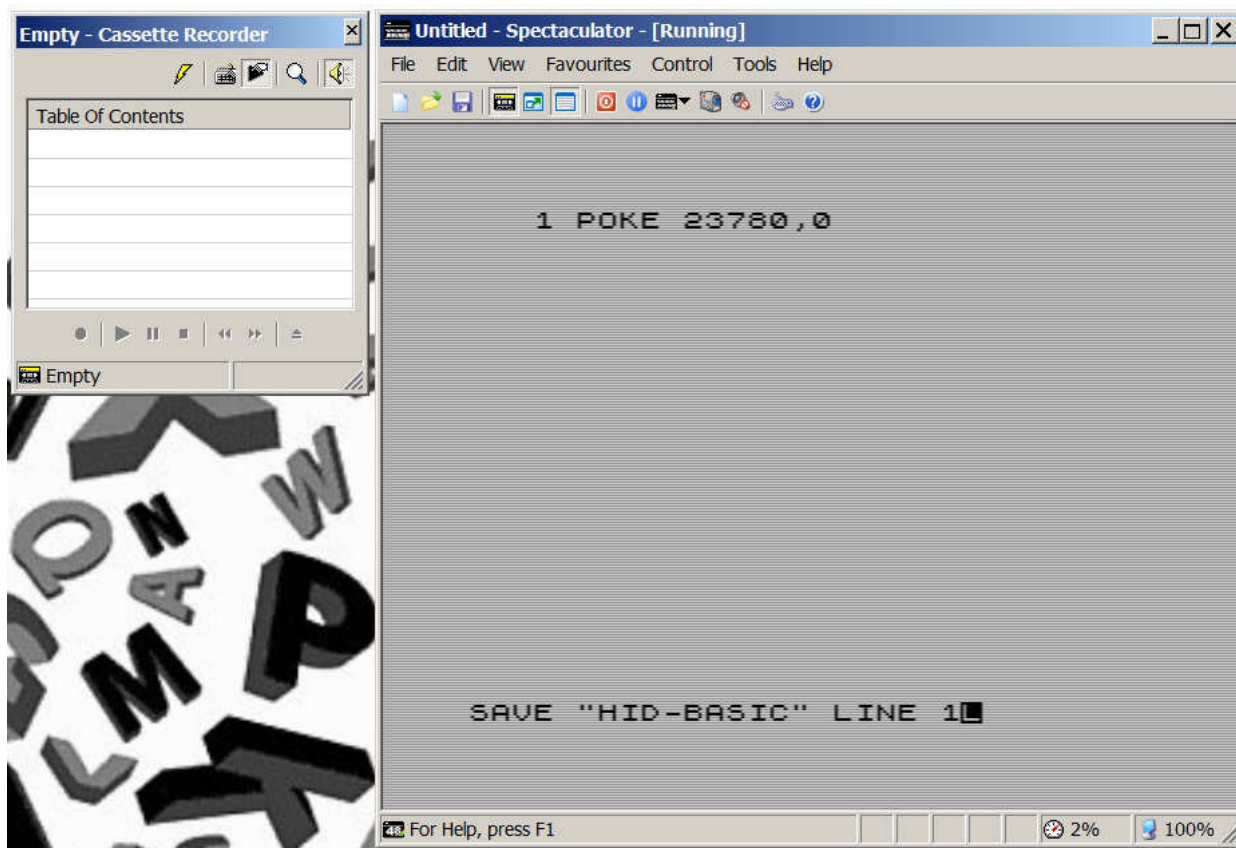


Рис. 2201. Подготовка программы для создания *.tzx перед сохранением в *.z80.

Запишите заготовку в формате *.z80, под именем «hid-basic» и закройте эмулятор. В качестве эксперимента, рассмотрим еще один метод создания файлов образа магнитной ленты. Попробуем создать *.tzx файл с помощью другого эмулятора.

Запускаем эмулятор Realspectrum, желательно настроенный на старт в режиме 48K. Перез запуском, для удобства, файл «hid-basic.z80» лучше поместить в корневой каталог эмулятора Realspectrum. Нажав клавишу F5, откроется окно «OPEN SNAPSHOT FILE». В нем необычно отобразится «HID-BASIC.Z80» большими буквами. Не обращайте на это внимание. Этот эмулятор не распознает маленькие буквы, и все пишет большими, но тем не менее названия записывает корректно:

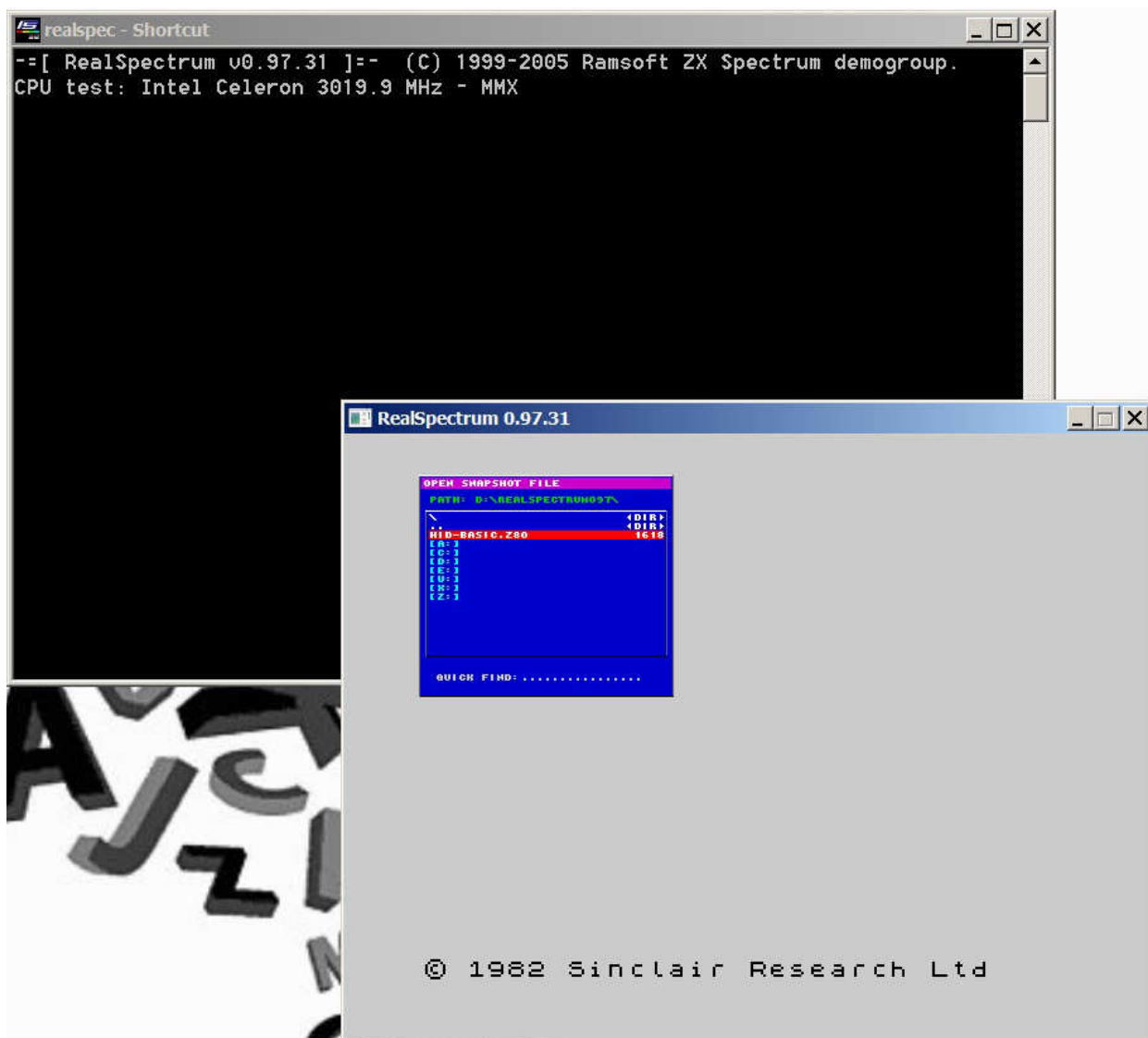


Рис. 2202. Открытие полуфабриката программы «hid-basic.z80» в эмуляторе RealSpectrum.

Курсорными клавишами выбираем «*HID-BASIC.Z80*», и открываем его. Появится программа с ожидающей ввода строкой, открывшись с того момента, на котором закончили с ней работать в эмуляторе Spectaculator, в момент сохранения в формате *.z80.

Ничего не делая в программе, нажимаем *ALT+F7* и выходим в окно «*TAPE OPTIONS*». Управляя маленьким треугольным курсором слева, стрелками на клавиатуре, выбираем меню «*OPEN TAPE FOR SAVING*». Нажав *ENTER*, выходим в подменю «*SAVE TAPE FILE*». Задаем имя выходному файлу на платформе PC в windows. Например, в поле «*FILE NAME*» введем «*hid-basic*». Вводим маленькими буквами, но в эмуляторе они отобразятся большими.

Выбираем формат файла, в каком будет записан блок. По умолчанию там стоит «*SAVE AS: TZX*», но стрелками мы можно выбрать 4 формата записи: *.tap, *.tZX, *.wav или *.csw. Сейчас в поле «*SAVE AS:*» оставим по умолчанию формат *.TZX:

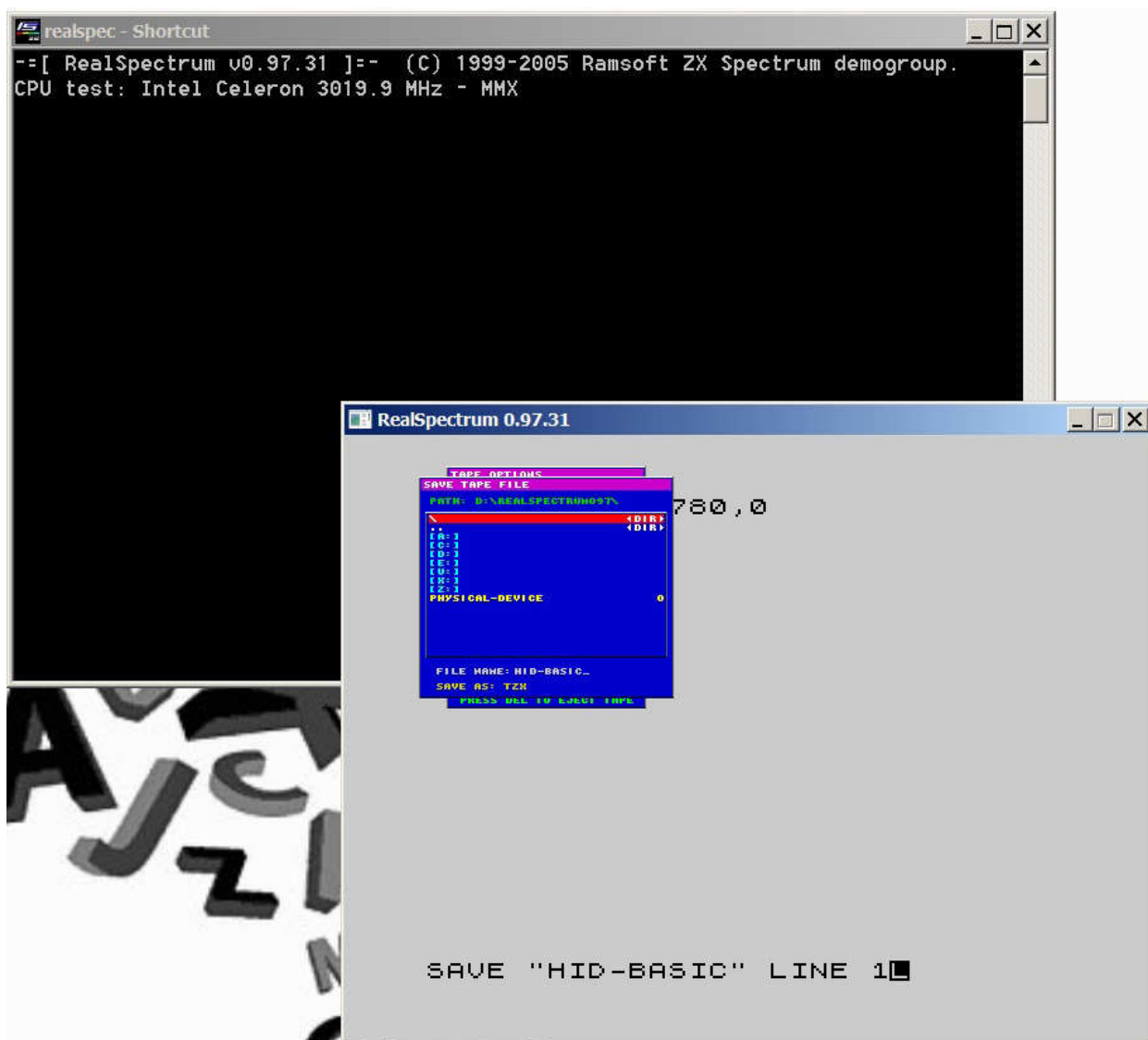


Рис. 2203. Окно сохранения файлов кассетного формата «SAVE TAPE FILE» эмулятора RealSpectrum

Нажимаем **ENTER**, и снова выходим в BASIC программу, ожидающую ввода строки. В своей программе на BASIC, наконец-то, вводим строку с мигающим курсором нажатием **ENTER**. Нижние строки очистятся, и на экране появится знакомая многим, надпись: `Start tape, then press any key. :`

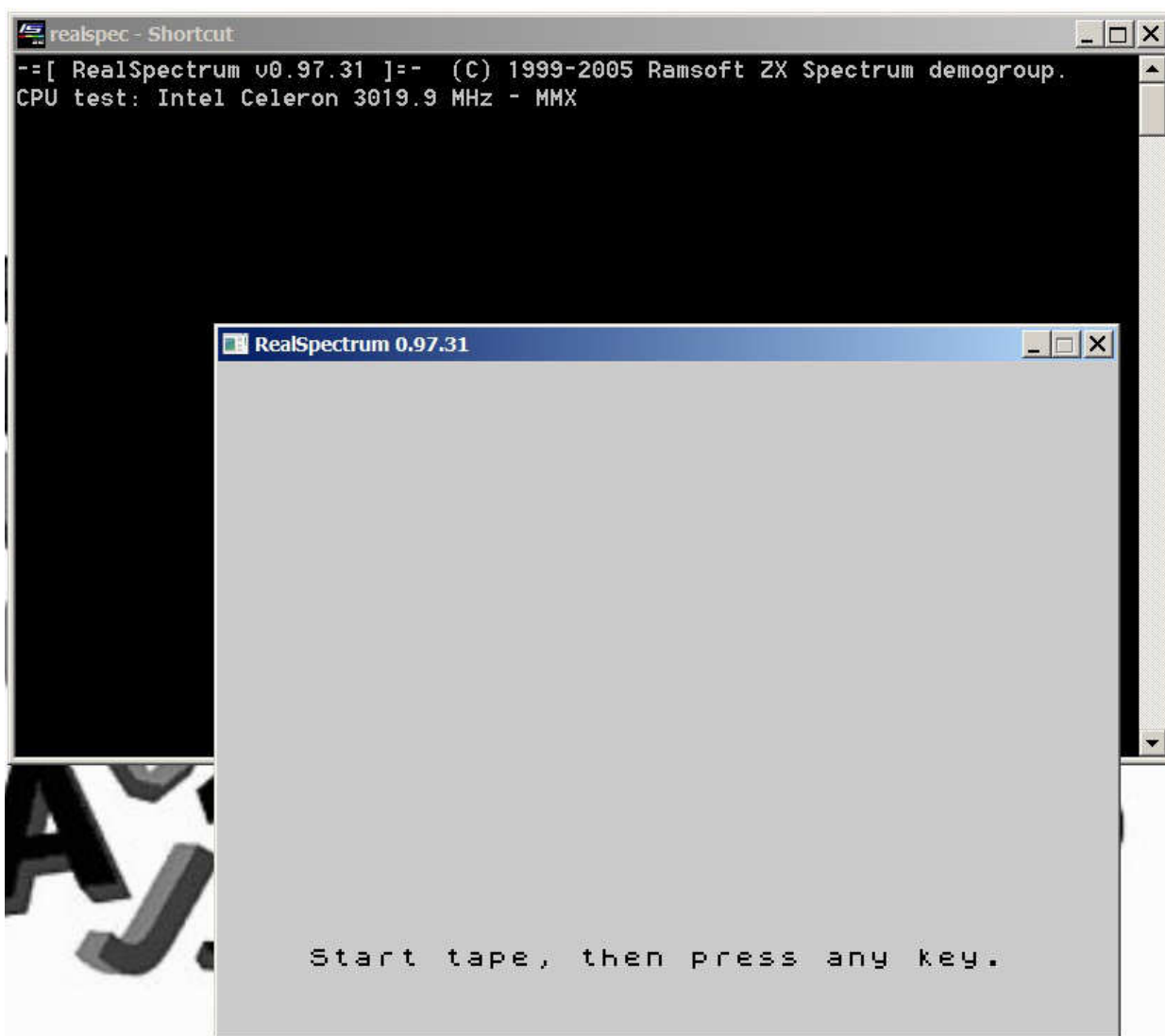


Рис. 2204. Ожидание начала записи файла в окне эмулятора RealSpectrum.

Нажмите **ENTER**, и дождитесь выполнения всей программы. Сообщение **OK**, **1**, просигнализируют об успешном выполнении программы в интерпретаторе BASIC. В режиме **.TAP* и **.TZX* запись блока произойдет без полос на рамке. Если вы записываете в **.WAV* или **.CSW*, запись будет производиться в реальном времени, желто-синими полосами, как на обычном Спектруме, с подсоединенным магнитофоном. Только на эмуляторе запись идет в файл.

Поскольку создавали файл формата **.tap* / **.tzx*, то после успешного завершения программы, требуется закрыть записываемый файл, если мы не собираемся добавлять еще блоки. Снова входим в меню «*TAPE OPTIONS*» по **ALT+F7**, треугольничком выбираем «*CLOSE HID-BASIC.TZX*»:

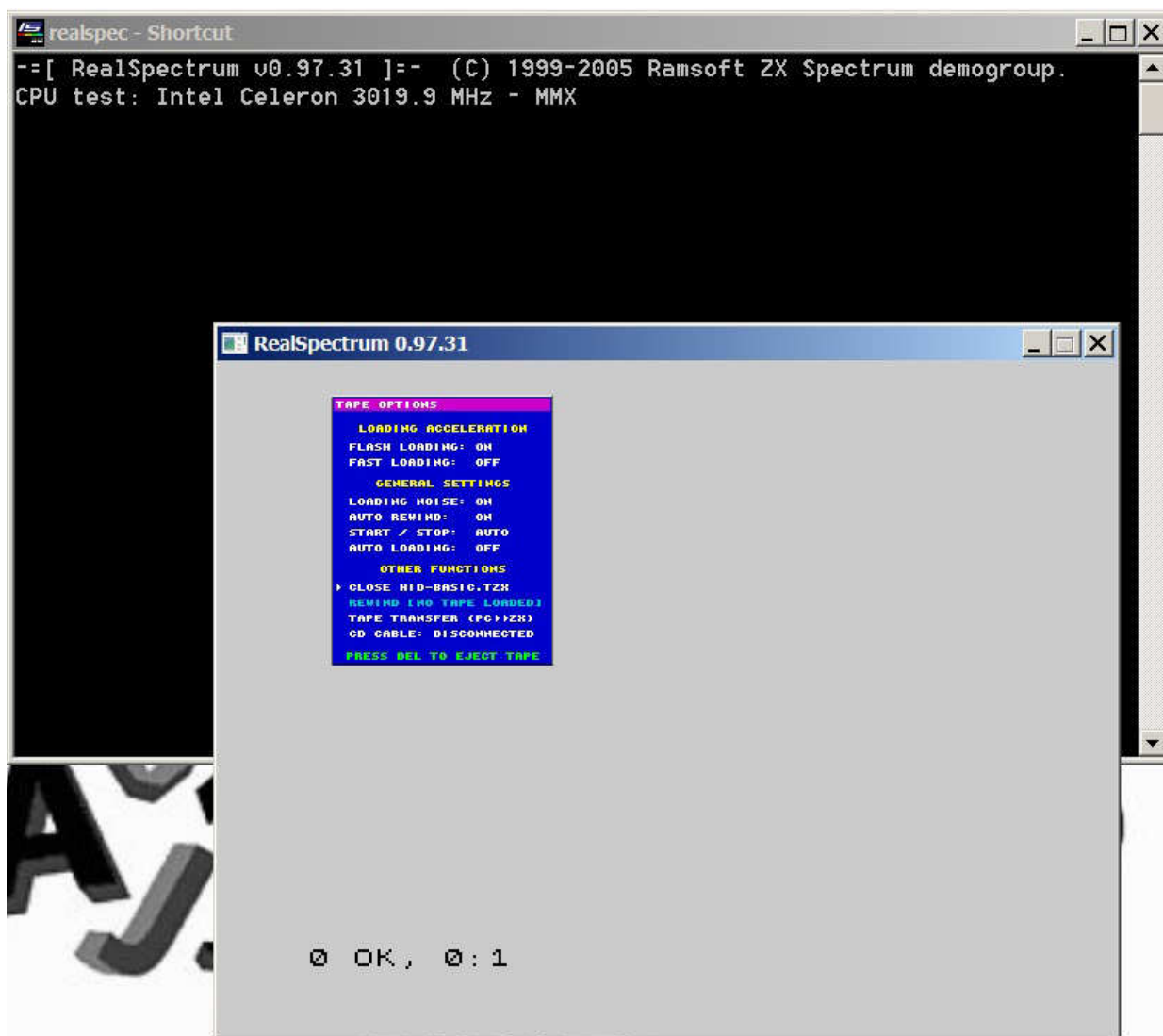


Рис. 2205. Меню «TAPE OPTIONS». Закрытие сформированного *.tzx файла. Треугольный курсор.

Нажмите *ENTER*, и ваш файл готов. В случае записи звукового файла *.wav или *.csw закрывать запись не требуется, она отключается автоматически, по окончании работы оператора *SAVE*. Теперь выключаем RealSpectrum нажатием клавиши *F10*. На вопрос «*REALLY [Y/N]*», подтверждаем выход клавишей «*Y*». Файл готов. Таким образом, мы рассмотрели еще один процесс создания файла образа магнитной ленты, только с помощью эмулятора Realspectrum. Он немного сложнее процесса создания файла в Spectaculator.

Теперь будем тестировать созданную программу, проверяя работоспособность. Откройте Spectaculator, настроенный на поддержку формата магнитофонной записи в реальном времени. Откройте окно виртуального магнитофона «*Cassete Recorder*», кнопкой на панели эмулятора или комбинацией *CTRL+K* (далее под словами *откройте Spectaculator*, будет подразумеваться открытие эмулятора с настроенными опциями магнитофона). Мышкой перетащите полученный файл образа магнитной ленты в окно магнитофона. В магнитофоне сразу появится информация о блоке и заголовке.

Для загрузки файла в окне эмулятора наберите команду *LOAD ""* *ENTER*, и мышью включите «*PLAY*» виртуального магнитофона. На экране начнется загрузка созданной программы:

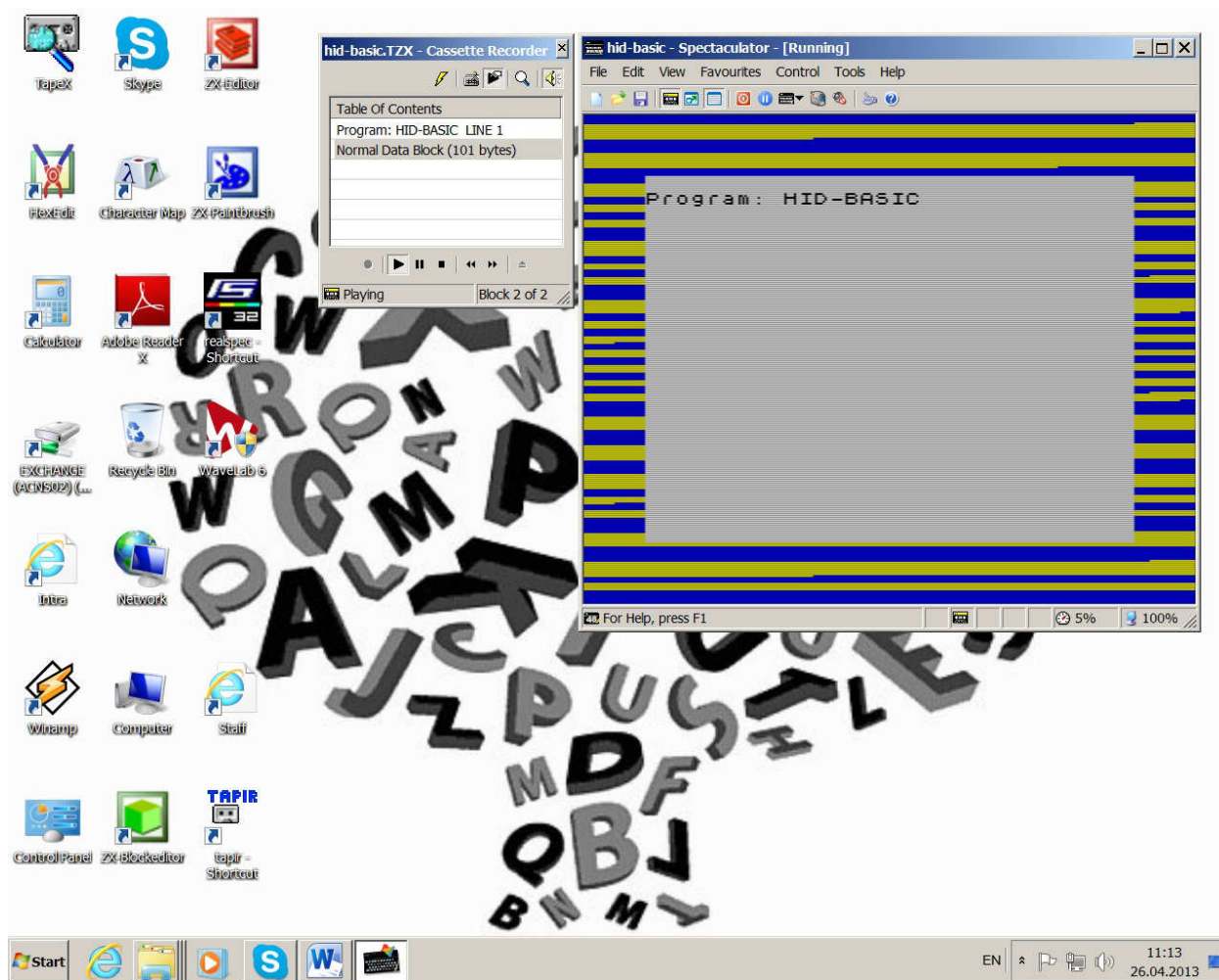


Рис. 2206. Процесс загрузки программы в эмуляторе на фоне Windows 7.

После загрузки блока «Program:», после автостарта выполнится строка 1 `POKE 23780, 0`, которая откроет «заглушку». Сразу проявятся две нижние строки. Интерпретатор, не успев выйти из программы, увидит их, и сразу начнет выполнять. На экране, под заголовком «Program:» выскочит надпись из строки 2: «Прогрейка работы скрытых строк программы». Строка 3 `POKE 23780, 255` вернет на место «заглушку», и скроет все строки, кроме первой.

Основной недостаток. Боится ввода новых строк. В случае ввода новой строки, происходит переразметка ОЗУ, изменяются некоторые переменные BASIC, и все невидимое стирается.

Глава 3.

Создание символьного массива, начиненного машинной программой.

Краткое содержание: Работа в EmulZwin, работа со встроенным Ассемблером, компиляция программы, отладчик *Debug Spectrum*, символьные массивы, создание многоблокового *.tzx файла.

В прошлой главе мы пробовали создать одиночный блок данных с заголовком. Теперь попробуем создать классическую двухблочковую программу. В такой программе сначала загружается блок «Program:», а затем «Bytes:». Но Bytes'ы мы еще будем создавать дальше, а сейчас попробуем немного изменить привычное представление о загрузке. В нашем случае загрузчик на BASIC, будет загружать символьный массив, в котором вместо символов содержится программа в машинных кодах.

Чтобы не набирать в Spectaculator, в область памяти по одной циферке машинного кода, попробуем написать простую программу на ассемблере, и скомпилировать с помощью эмулятора EmulzWin, Владимира Кладова, в произвольную область памяти. Заодно познакомимся с этим эмулятором, потренируемся в нем работать, и вкратце изучим его преимущества.

Если работаете под Windows 7, версию 2.7 необходимо запускать в режиме эмуляции XP, более ранние версии работают без этого режима, но могут конфликтовать с программами из пакета Office и, особенно, Internet Explorer. Поэтому параллельная работа IE крайне нежелательна.

Итак, откройте эмулятор EmulzWin (2.5 или 2.7), настроенный в режиме 48K. Наберите в нем следующую программу:

```
1 LOAD ""DATA а$(): RANDOMIZE USR 24000
```

Введите строку в эмулятор. Теперь создадим «начинку» для строкового массива. Для этого нажмите кнопку «Tools» на панели эмулятора, и в открывшемся меню выберите «Assembler». Рядом с эмулятором откроется дополнительное окно «ZX Assembler++»:

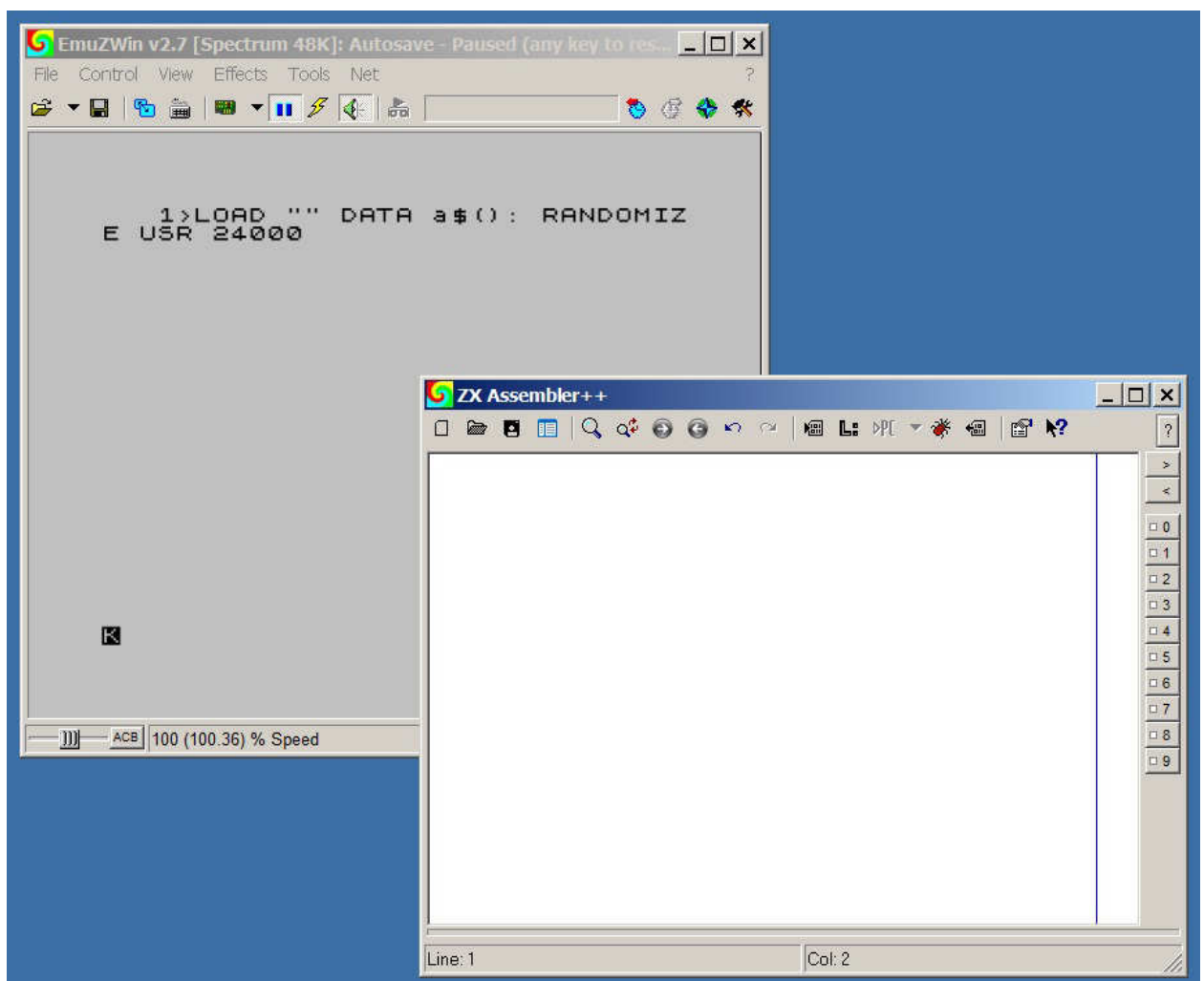


Рис. 2300. Эмулятор EmulZWin с набранной программой и открытым окном ассемблера (ZX Assembler++)

Теперь перейдем в окно ассемблера, и наберем короткую машинную программу, которая в качестве теста, выведет красную букву «А» на экран. Чтобы подсчитать точное количество байт, оттранслируем ее в произвольное место памяти, например в 40000:

```
ORG 40000
```

```
LD A, 02
CALL 5633
LD A, 16
RST 16
LD A, 2
RST 16
LD A, 65
RST 16
RET
```

Чтобы не набирать руками, просто выделите эту программу мышкой прямо отсюда (для .pdf и .doc формата), и скопируйте в буфер, нажав *CTRL+C*. Перейдите в окно ассемблера и нажмите там *CTRL+V*. Это и есть основное достоинство EmulZWin, когда можно дизассемблировать и компилировать программы, включающие в себя целые игры на десятки килобайт, простым переносом в любой редактор под windows. Отредактировав данные, блок можно вставить снова прямо во время работы программы, в режиме реального времени.

Теперь в окне ассемблера на панели, нажмите кнопочку «*COMPILE*». Почерневшая кнопочка «*PC*», просигнализирует об успешной компиляции программы:

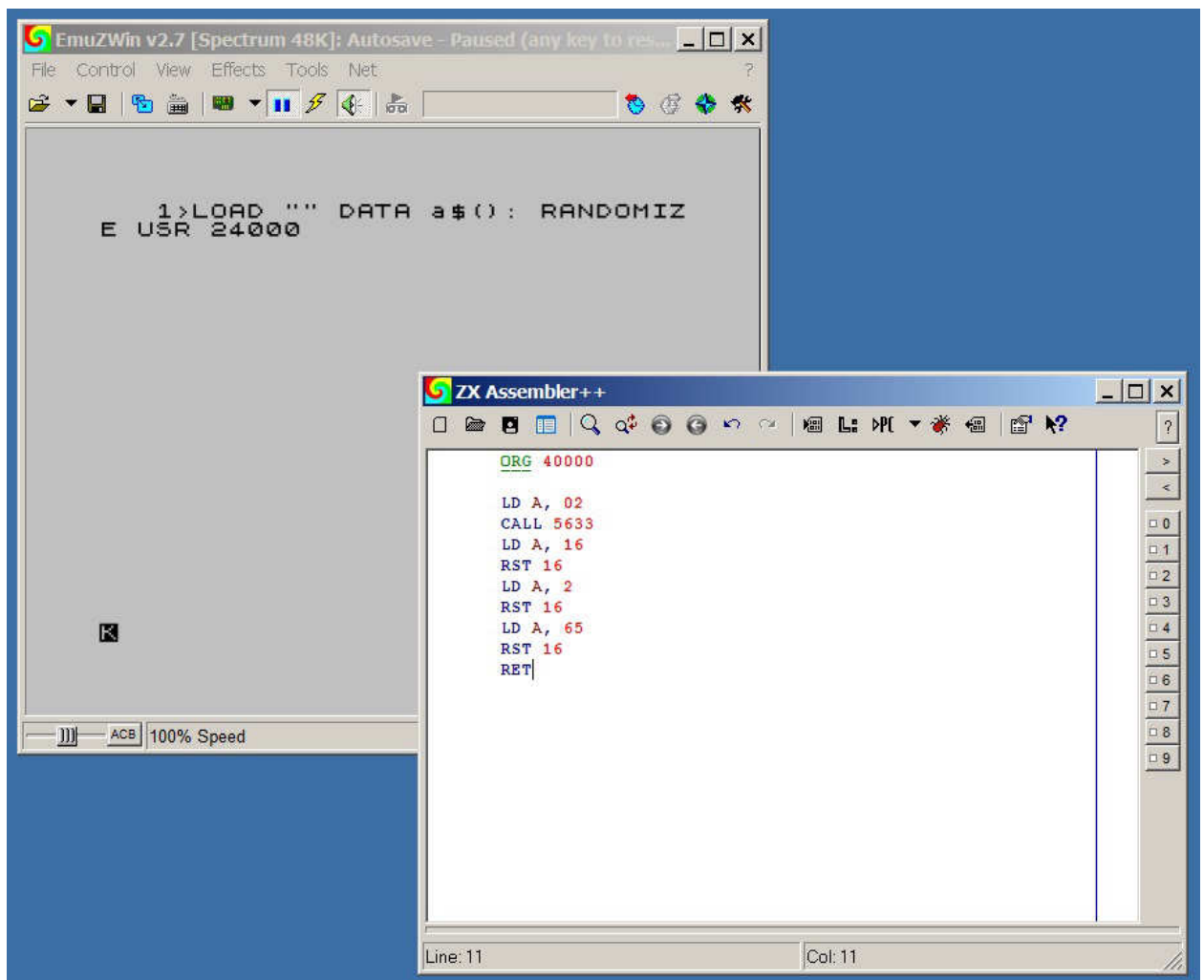


Рис. 2301. Компиляция тестовой программы на ассемблере Z80 в эмуляторе EmulZwin.

Если кнопочка «*PC*» осталась серой, значит программа не скомпилировалась. Ищите ошибку в тексте программы. Настоятельно не рекомендуется вводить в

компилятор русские буквы. Это может привести к ошибке, и внезапному закрытию приложения. К сожалению, версии эмулятора очень сырые, поэтому внимательным образом следите за вводимой информацией.

После успешной компиляции программы, перейдите в окно эмулятора, не закрывая ассемблер, и проверьте, набрав команду:

```
RANDOMIZE USA 40000
```

В левом углу экрана виртуального Спектрума загорится красная буква «А». В машинных кодах выполнена программа, эквивалентная строке BASIC:

```
PRINT INK 2; "A"
```

Теперь снова выходим в окно ассемблера, и нажимаем кнопку с жучком (*DEBUGGER*). Откроется окно многофункционального отладчика. Для удобства просмотра на панели отладчика нажмите кнопку с пиктограммой 10 (*Decimal View*) и все адреса с данными отобразятся в десятичном формате. Синим цветом отмечены системные метки в ПЗУ и ОЗУ:

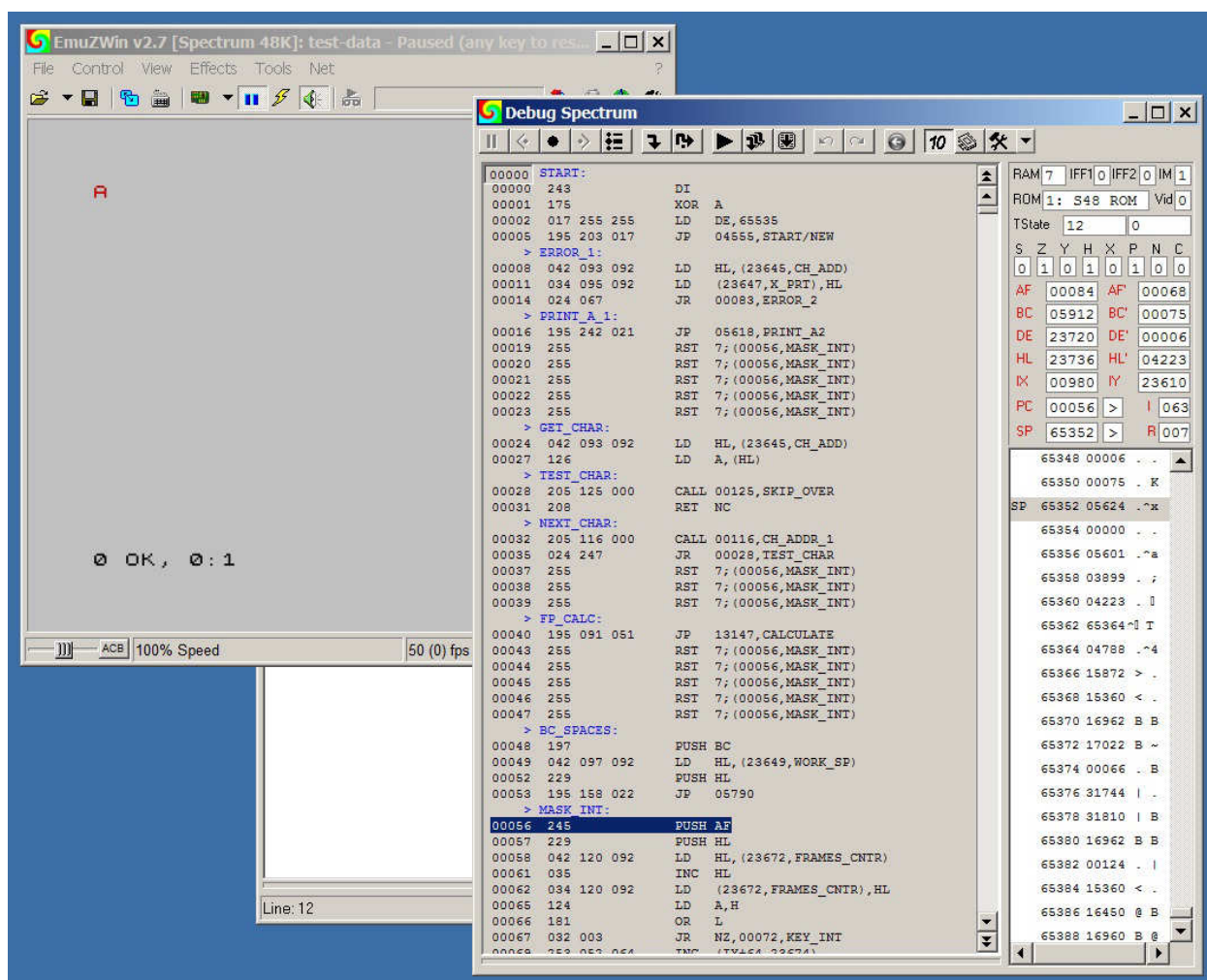


Рис. 2302. Эмулятор EmuZWin. Окно «Debug Spectrum». ПЗУ и системные метки.

Прокручиваем область памяти до адреса 40000 и находим скомпилированную программу. Видим, что она занимает в памяти всего 15 байт:

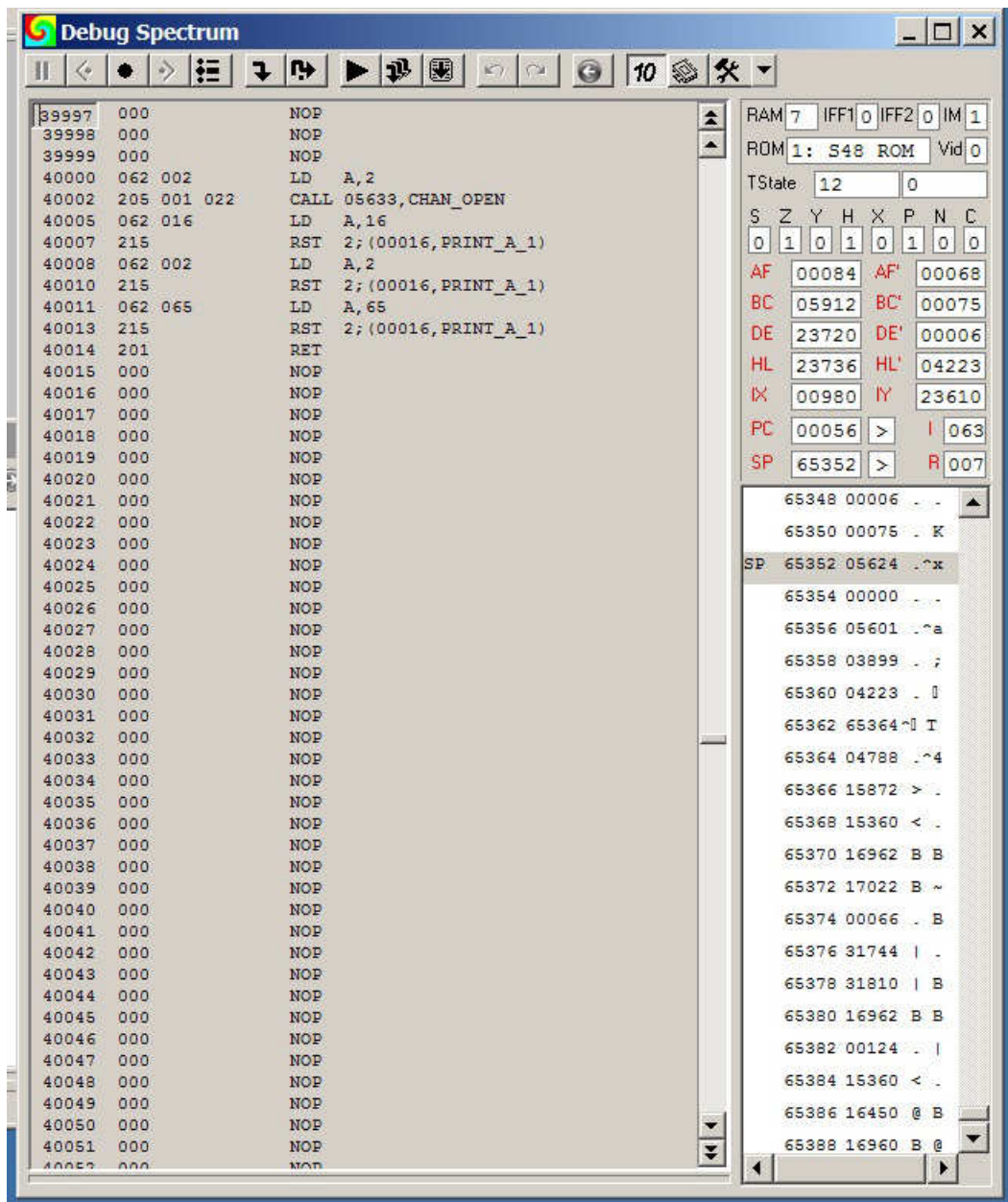


Рис. 2303. Скомпилированная программа в памяти эмулятора EmulZWin.

Теперь, зная длину программы, переходим в окно эмулятора. Набираем и вводим строку с массивом, состоящим из 15-ти букв «q»:

```
LET a$="qqqqqqqqqqqqqqq"
```

Можно проверить работоспособность массива. Для этого наберите `PRINT a$`. Ни в коем случае не вводите операторы `RUN` или `CLEAR`, которые сотрут этот массив из памяти. Если понадобилось перед записью проверить программу, используйте только `GOTO`.

Переходим в окно «*Debug Spectrum*», и ищем переменную **VARS** (23627 и 23628), которая укажет на адрес начала переменных бейсика. В эмуляторе Владимира Кладова она промаркирована как «**SYS VARS**». В ней мы увидим значения 230 и 92. Следовательно, массив будет располагаться по адресу: $92 \cdot 256 + 230 = 23782$.

Проматываем строки с адресами до нужного, и видим, заветную строку с 15-ю буквами «q», ограниченные в конце маркером «128». Располагаются переменные сразу после строк программы:

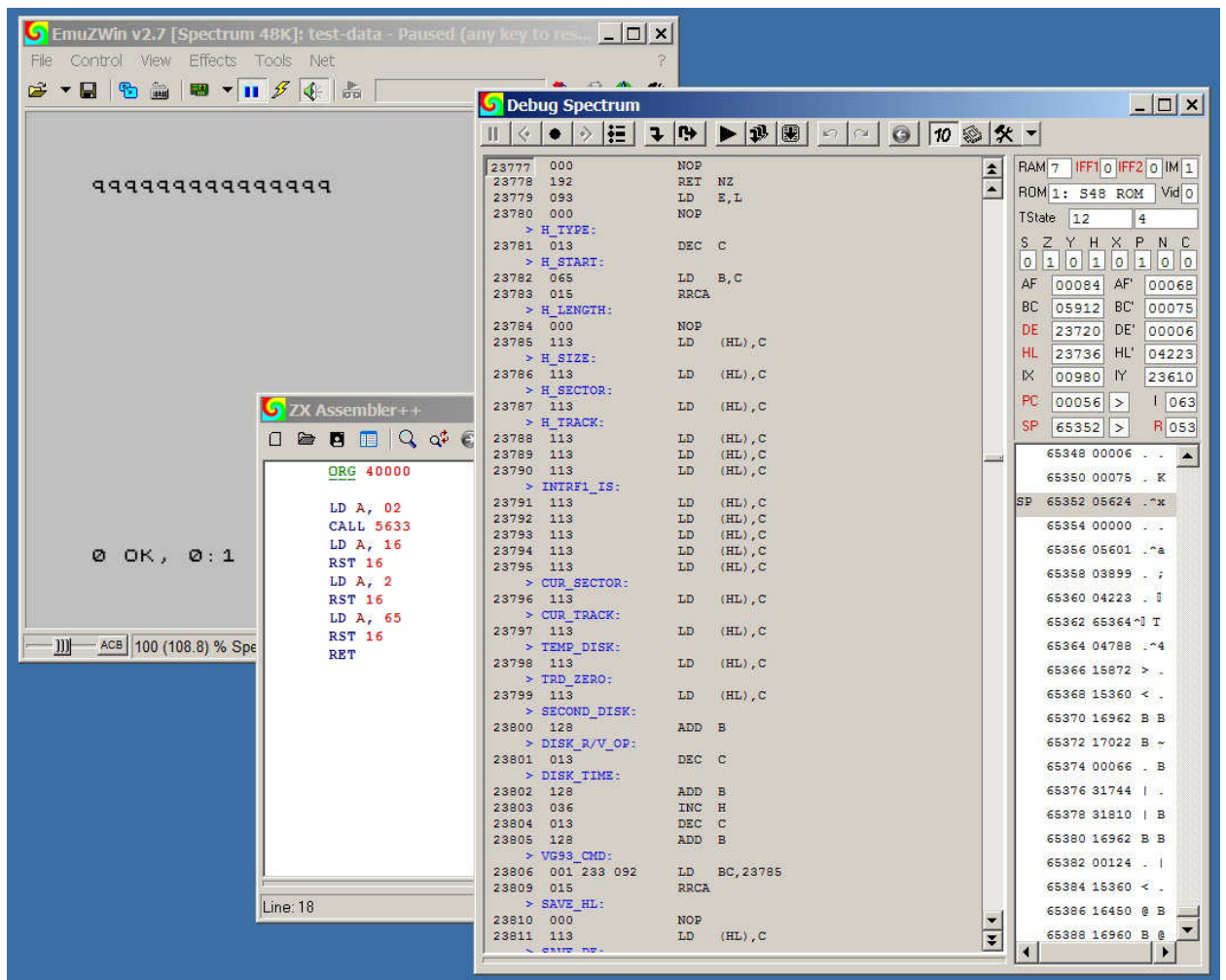


Рис. 2304. Окно *DEBUG SPECTRUM* эмулятора *EmilZWin*. Расположение массива *a\$* в памяти.

В этом эмуляторе могут сбить с толку заголовки системных переменных режима 128K с дисководом. Просто не обращайтесь на них внимания. Первым по указанному адресу стоит число 65, эквивалентное букве «a» массива, затем 2 байта длины, а с адреса 23785 сам массив из букв «q» (число 113). Следовательно, вставлять свою программу будем с адреса 23785. Переходим в окно «ZX Assembler++», и в директиве **ORG**, адрес 40000 исправляем на 23785. Затем выходим в окно эмулятора и редактируем строку 1, после **RANDOMIZE** **USR** исправив с 24000 на адрес 23785. «Украсить» программу можно нулевой строкой, набрав и введя: **POKE 23756,0**

Теперь в эмуляторе наберем подготовительную строку для записи будущего блока:

```
SAVE "TEST-DATA" LINE 0: SAVE "test-data"
DATA a$( )
```

Перейдем в окно ассемблера, и скомпилируем программу в новый адрес, нажав кнопку «Compile». Переходим в окно отладчика «Debug Spectrum», и окончательно проверяем свою программу. Должно получиться примерно следующее:

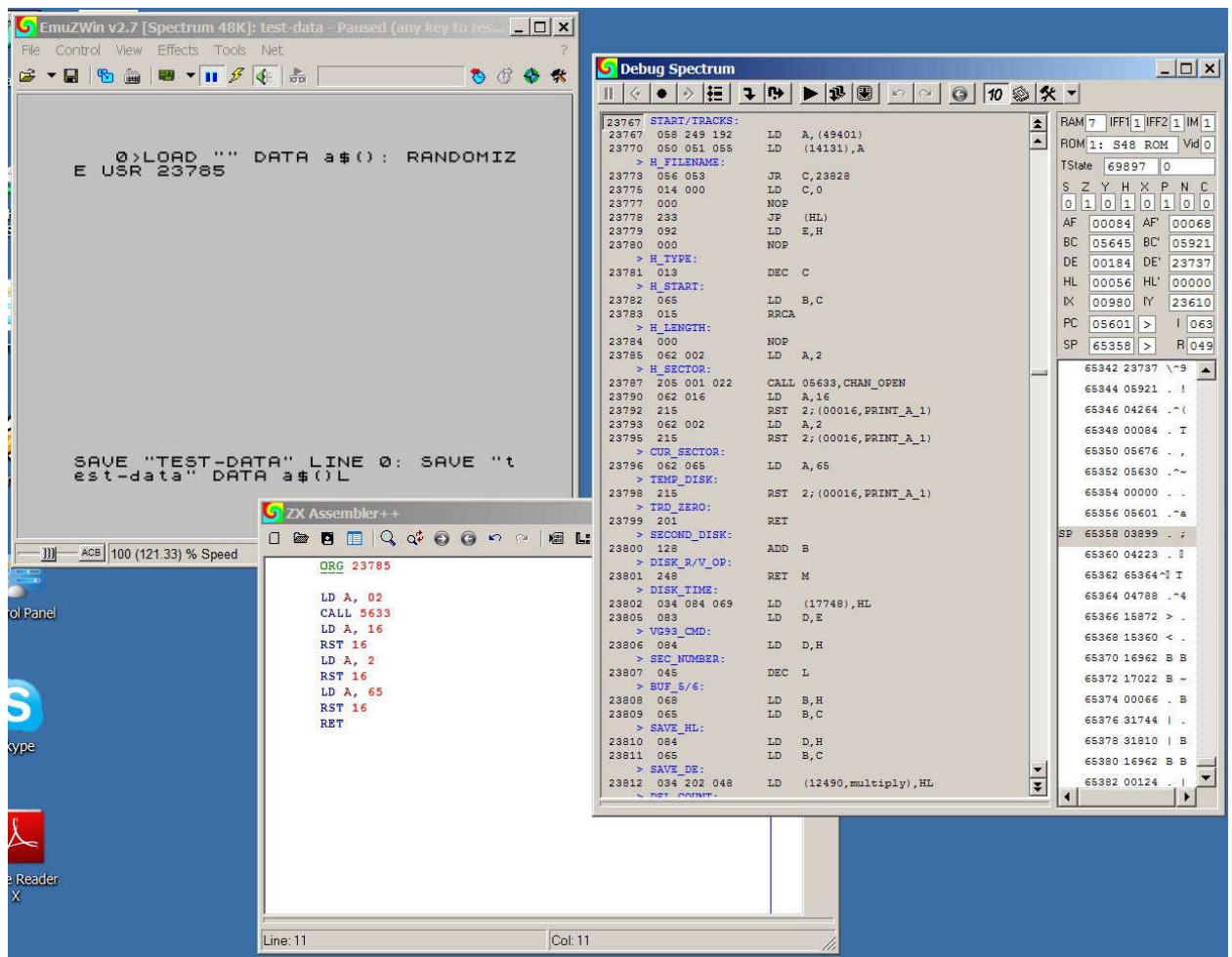


Рис. 2305. Эмулятор, ассемблер и отладчик. Окончательная проверка скомпилированной программы.

Закрываем окна отладчика и ассемблера. Переходим в окно эмулятора, чтобы готовый «полуфабрикат» сохранить в виде слепка памяти *.z80. На панели эмулятора нажимаем кнопку «File», а в открывшемся меню «Save As». Откроется окно «Select File To Save State». Из выпадающего списка в «Save as type:» выбираем «Z80 raw snap format (.z80)», а в строке «File name:» пишем «test-data»:

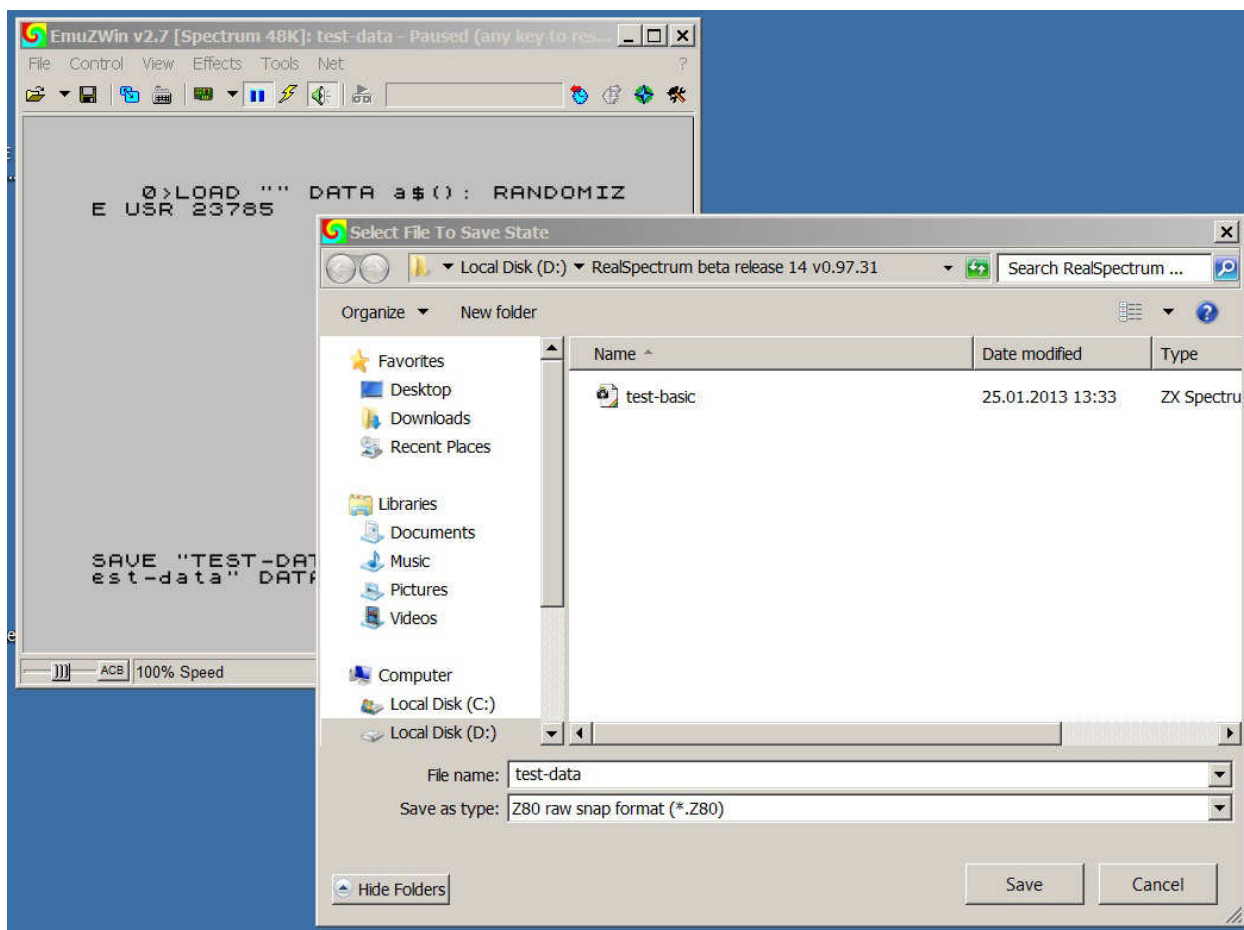


Рис. 2306. Сохранение готовой программы слепком памяти *.z80 в эмуляторе EmuZWin.

Сохраняем файл и выходим из эмулятора. Полуфабрикат готов. Еще раз убеждаемся, что *.z80 самый удобный и универсальный формат, который поддерживают практически все эмуляторы.

Теперь создадим из него *.tzx или *.tap файл. Для этого воспользуемся любым из способов, описанных выше, в главах 1 и 2. Задайте имя готовому файлу «test-data». Отличие будет в том, что теперь готовый *.tzx файл будет содержать два блока данных с заголовками.

При записи файла надпись (Start tape, then press any key) выскочит дважды. Сначала для блока «Program:», а потом для «Character array:». Если вы записывали файл с помощью Spectaculator, то закрывать файл (по ALT+F7) следует только после записи обоих блоков.

Теперь остается самое интересное. Запускаем Spectaculator, заряжаем собранную программу в магнитофон, нажимаем LOAD ""ENTER, и наблюдаем за загрузкой. Заметим, что теперь в виртуальном магнитофоне лежат два блока данных, в виде 4-х строк. (Если вы создавали *.tzx с помощью Spectaculator, то сверху может присутствовать и информационный тег «[Created by Spectaculator]»). По мере загрузки программы, курсор передвигается по блокам сверху вниз:

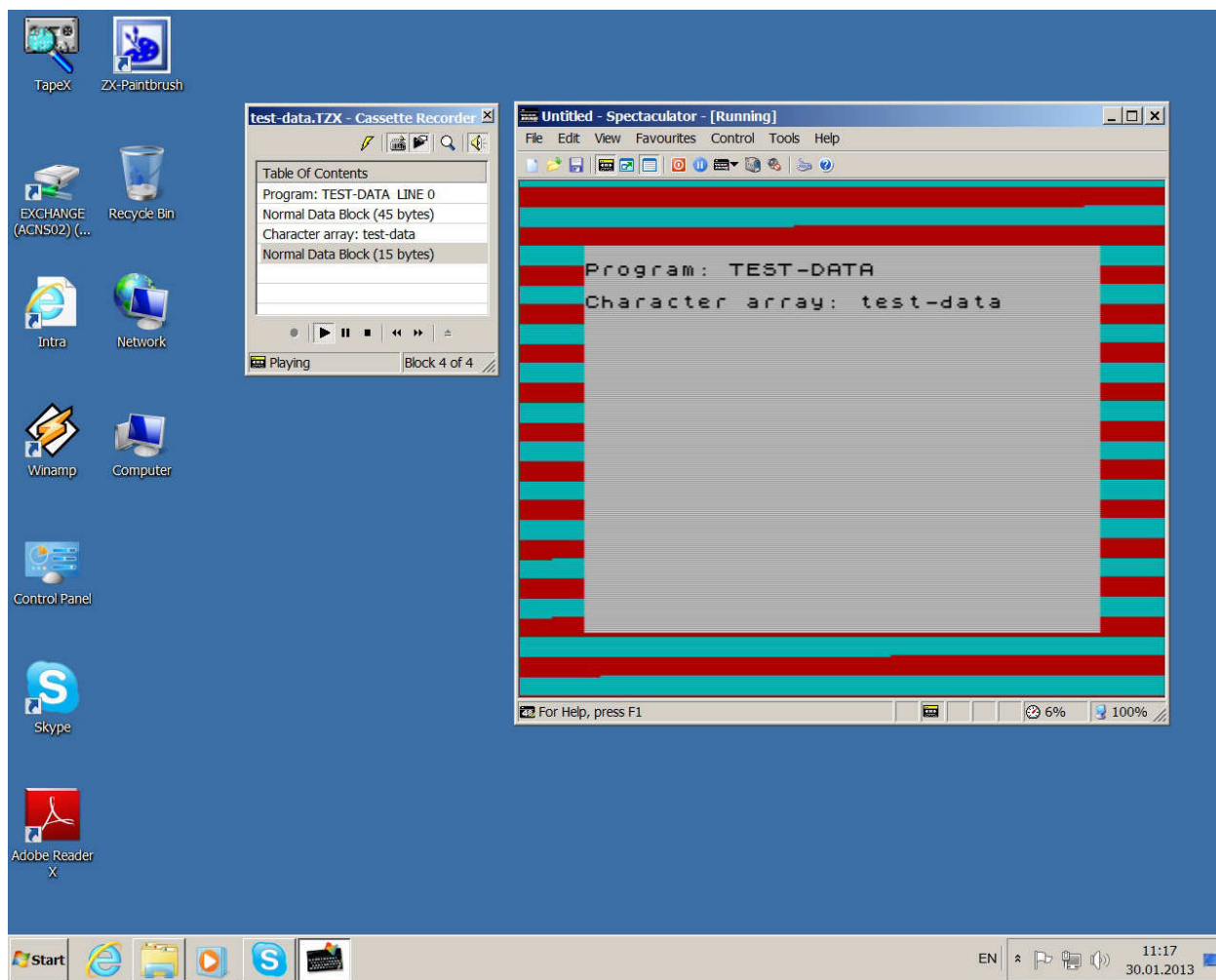


Рис. 2307. Двухблоковая программа в процессе загрузки на эмуляторе Realspectrum.

Если вы сделали все правильно, то после окончания загрузки под названиями блоков выскочит красная буква «A», которая покажет, что программа в машинных кодах выполнена из загруженного символьного массива.

Глава 4.

Запись автостартующего блока «Bytes:» с BASIC-строкой из машинного кода.

Краткое содержание: машинные коды в строке BASIC, автозапуск программы в машинных кодах из вводимой строки, RANDOMIZE USR 0.

Многие знают, как во многих играх по LOAD "", загружается BASIC программа, начинается с нулевой строки, и состоит из:

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

или еще лучше:

```
0 CLOSE# USR 0 ...
```

Следом за USR 0 стоит куча всякого «мусора», скрытого под нечитаемыми управляющими символами. Например, вот так:

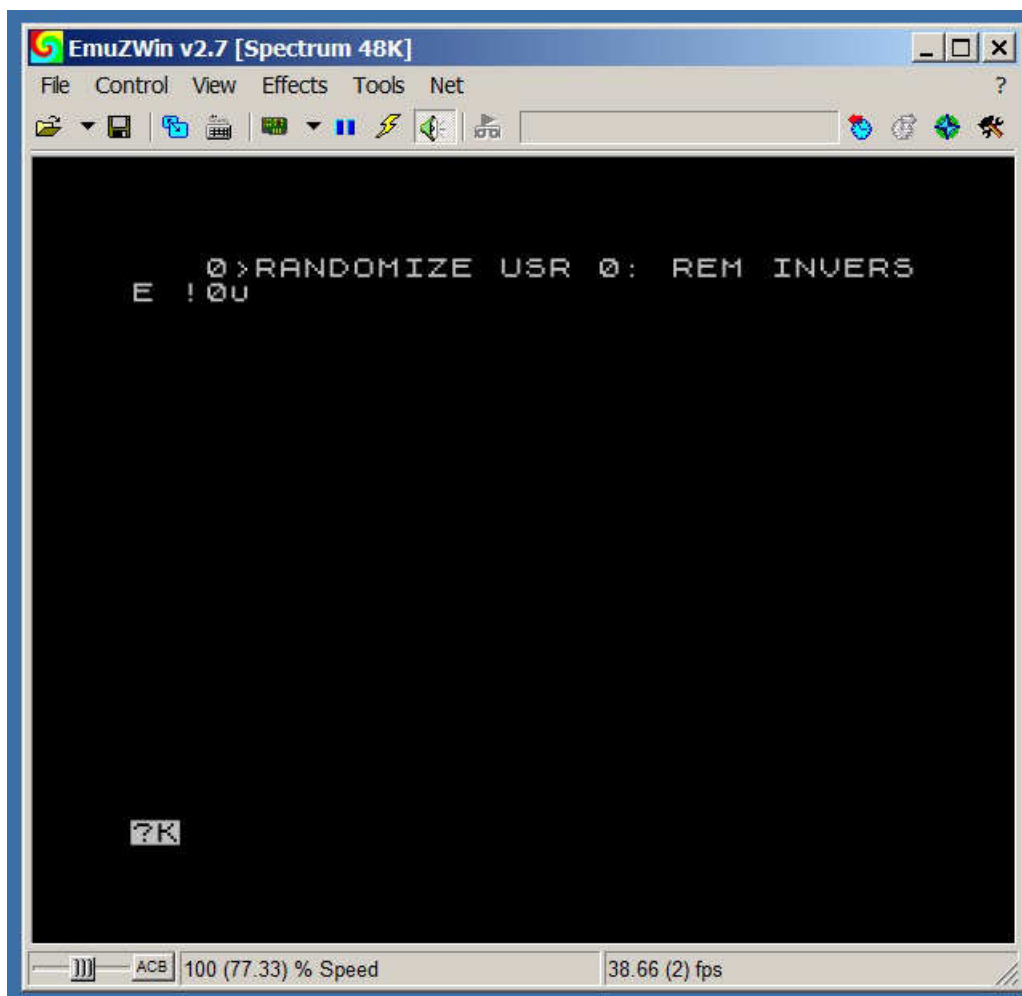


Рис. 2400. Типовая «Программа с помойкой», где в строке BASIC записана программа в машинном коде.

Все это часто оформляется черной рамкой с черным экраном, а внизу мигает вопросительный знак с курсором. В детстве, когда только начал изучать BASIC, не совсем понимая смысл, я называл такие блоки данных «Программа с помойкой». Я предполагал, что эта самая «помойка» сделана для устрашения.

В старых книжках о методах защиты, данные вещи неоднократно описывались. Таким образом машинная программа-загрузчик записывается в область BASIC, а запускается по команде `RANDOMIZE USR 0` [мнимый ноль], где в 3-м и 4-м байте представления числа «0» записан настоящий адрес перехода. (в 4-м старший байт*256, в 3-м младший)

За таким загрузчиком, как правило, следует ожидать какой-то нестандартный метод считывания. Как минимум, после такой «Программы с помойкой» пойдет загрузка без заголовка, или полосками других цветов. Но бывают и более экзотические виды, когда загрузка идет без полос на рамке, или с параллельным выводом текстовой информации на экран (невзломанный DEFLEKTOR). Также после такой программы могут идти блоки турбозагрузки. Изучением структуры сигналов будем заниматься позже.

Кстате говоря, вместо `RANDOMIZE`, перед `USR` можно поставить некоторые другие команды, требующие после себя переменную. Например: `PAPER USR 0`, `BORDER USR 0`, `BEEP USR 0`, и так далее. Главное, чтобы «скормить» `USR 0`, и заставить его выполняться, а далее уже запускается машинная программа, игнорируя переменные, требующиеся для этих команд в BASIC-режиме.

А вот после выполнения такой машинной программы, если предусмотрен выход в BASIC, то задним числом получаем сообщение об ошибках и некоторые глюки. Второй

раз программа с этого адреса уже не запустится. Все области бейсик системы сместятся на несколько десятков байт вниз.

Такой метод можно использовать для одноразового использования, когда после запуска загрузится машинная программа, выход из которой осуществляется только через кнопку *RESET*.

Некоторые команды, в сочетании с *USR 0*, ведут себя более корректно. Например: *CLOSE #USR 0* и *GO TO USR 0* можно запустить повторно. Но в любом случае происходит смещение BASIC программы вниз, за область ввода строки, а используемая ранее область становится заброшенной и теперь бездействует.

Почему *USR 0*, а не *USR 23759*, например? В старые времена этот метод применялся в качестве защиты, чтобы сбить с толку начинающего любителя взламывать игры. Ведь если такую строку взять на редактирование, а затем снова ввести, то значения обнулятся, и ноль станет нулем. Потребуется снова туда записывать значения, чтобы при запуске программы не произошел настоящий переход по адресу 0, то есть сброс.

Сейчас ни о какой «защите» речь не идет. Защиты утратили свою актуальность еще задолго до массового появления IBM совместимых компьютеров, в качестве домашних. Еще в 1991 году в 1-м издании книги «ZX-Spectrum для пользователей и программистов» Ларченко и Родионова, в главе «Обзор программного обеспечения» проскочила такая фраза:

«...Причем если раньше для «взломывания» программы требовалась масса изобретательности и таланта, то теперь, после появления так называемых мультифейсов (Multiface), и других устройств, позволяющих нажатием одной кнопки записать содержимое всей памяти на ленту или диск, снять защиту может и первоклассник...»

С первоклассниками, конечно, погорячились. Все-таки требуются некоторые знания машинных языков даже сейчас. Но, тем не менее, если уж в 1991 году, еще до эпохи эмуляторов, уже вскрывать защиты стало вполне реально, то спустя более 20 лет, в эпоху эмуляторов, сделать это в десятки раз легче, чем тогда с помощью мультифейса. Сейчас пара манипуляций мышью в эмуляторе, и на рабочем столе windows лежит текстовый файл с дизассемблированным фрагментом программы, включая метки и переходы.

«Ноль» (1, 2, 3...) очень удобно использовать в программной строке для экономии байтов. Ведь число «23759», по сравнению с «0», отодвинет адрес строки еще на 4 байта вниз.

Давайте, усовершенствуем программу, и пойдем еще дальше. Попробуем обойтись без *USR 0* вообще. Тогда в BASIC строке останется чистый машинный код без вспомогательных команд. Откроем эмулятор EmulZWin, затем в нем окно «ZX Assembler++», и напишем (или скопируем в его окно) следующую программу:

```
ORG 30000
```

```
LD A, 02  
CALL 5633  
LD A, 16  
RST 16  
LD A, 1  
RST 16  
LD A, 84  
RST 16  
LD A, 101  
RST 16  
LD A, 115
```

```
RST 16
LD A, 116
RST 16
RET
```

С помощью отладчика SPECTRUM DEBUGGER определяем ее длину, а это 24 байта. Теперь в самом эмуляторе наберем и введем строку:

```
1 REM  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

После `REM` печатаем 23 буквы «`a`», зарезервировав место под будущую программу. Затем открываем окно отладчика, и выясняем, что буквы «`a`» расположились в памяти с адреса 23760, а команда `REM` стоит в ячейке 23759. Следовательно, мы подготовили 24 байта (23+1). В окне ассемблера исправляем «`ORG 30000`» на «`ORG 23759`», но пока ничего не компилируем.

Теперь украшаем программу «нулевой строкой», и вводим `FOKE 23756,0`. Убеждаемся, что строка стала нулевой и системные области не сдвинуты. Затем вводим подготовительную строку:

```

      SAVE "supercode" CODE 23613,300:
RANDOMIZE USR 23759

```

А дальше действуем по методике, описанной в 1-й главе этой части книги. В отладчике находим конец редактируемой строки. В данном случае он будет 23816. Временно сохраняем программу в формате .z80 под именем «*supercode.z80*» Теперь, как описывалось раньше, узнаем «индекс смещения» после выполнения программы, прокрутив сценарий развития до конца. Для этого компилируем программу в окне Ассемблера, а в эмуляторе вводим свою строку, и терпеливо ждем выполнения всех операторов BASIC до конца. Открываем «*Debugger*» и видим, что программа сползла на адрес 23834. Вычисляем длину $23835 - 23513 = 222$ байта.

Снова загружаем слепок памяти в эмулятор, возвращаясь к ожиданию записи. В окне ассемблера компилируем программу. В своей программе, подкорректируем длину блока с 300, на 222 байта, и окончательно сохраняем полуфабрикат в .z80 под именем «*supercode*»:

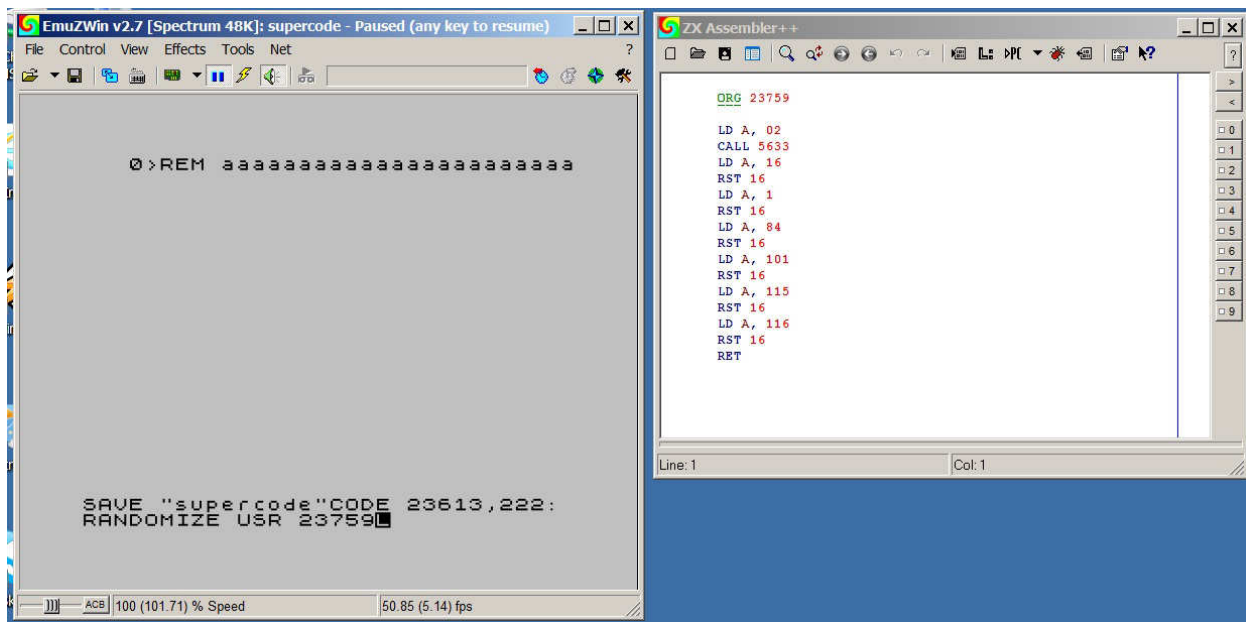


Рис. 2401. Подготовленный полуфабрикат программы «supercode.z80» на эмуляторе EmulZWin.

Создайте файл «*supercode.tzx*» удобным вам способом.

Загрузку в память будем производить по команде `LOAD ""CODE`. После загрузки, если вы все правильно сделали над надписью «`Bytes : supercode`» выскочит синяя надпись «`Test`». Это означает, что машинная программа автоматически запустилась. При этом программа вышла в интерпретатор BASIC без повреждений, с выдачей сообщения `OK, 0:2`:

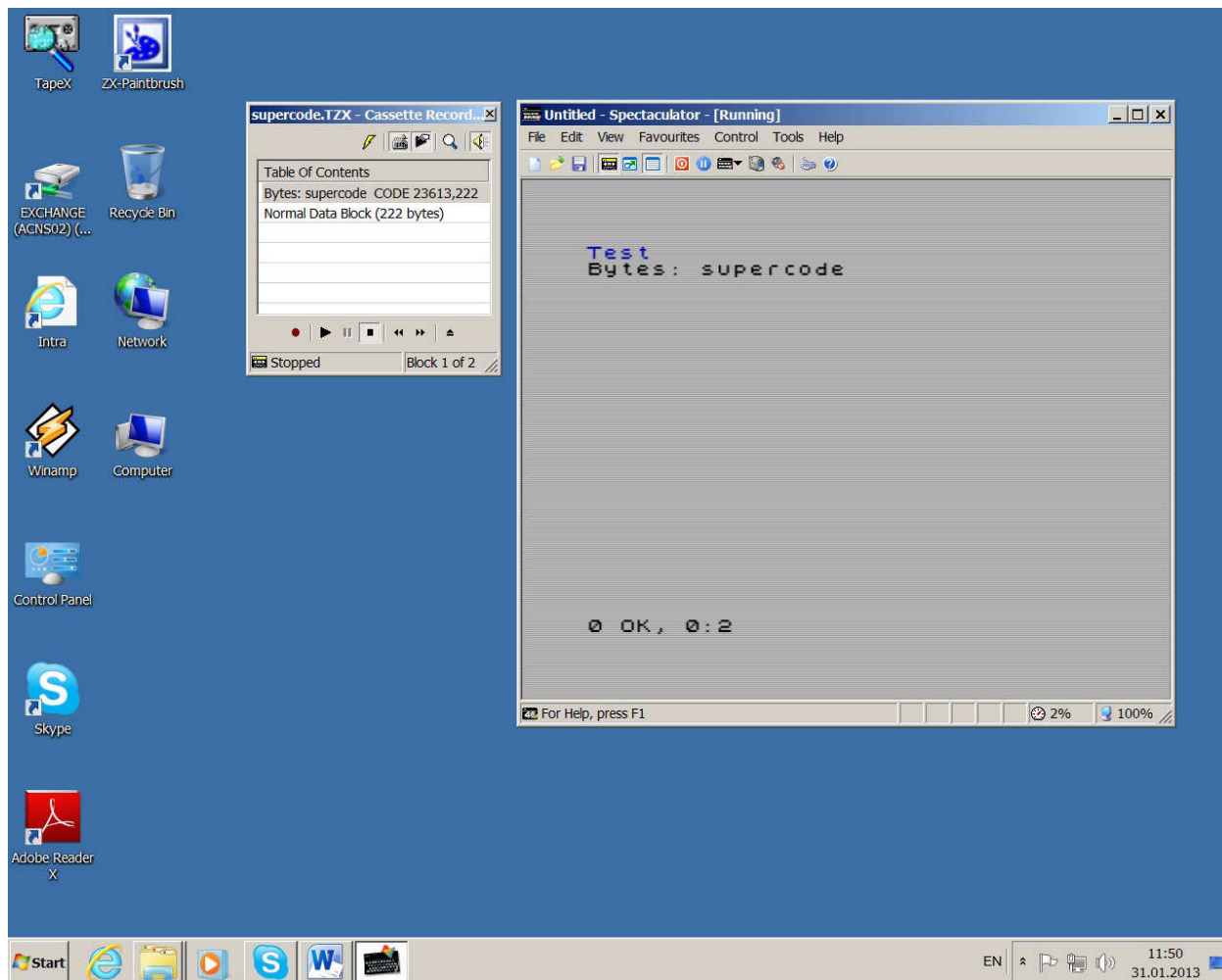


Рис. 2402. Программа «*supercode*» после выполнения и корректного выхода в BASIC.

Если вы откроете «*Debugger*», и посмотрите содержимое памяти с адреса 23755, то увидите, что два байта номера строки, плюс два байта длины, переведенные в ассемблер, выглядят таким образом:

```
NOP
NOP
ADD HL, DE
NOP
```

Из номера+длины строки получилась вполне нейтральная и безопасная программа. Ее выполнение не повлияет на основную программу. В нашем случае, стартовый адрес машинной программы может равняться значению переменной **PROG**, а это, в неискаженном варианте, 23755.

Таким образом, вы можете повторно запустить свою программу, набрав команду `RANDOMIZE USR 23755`.

В этой программе получилось очень интересное сочетание недокументированных особенностей. Программа в машинных кодах видна как строка в BASIC, при этом не имеет ни единого оператора BASIC. Она записана блоком «`Byte` : », но, тем не менее, автоматически запускается.

Можно подобрать и другие безопасные значения для длины строки. С номером строки можно ничего не делать. Двухбайтовый номер нулевой строки всегда будет выглядеть как две безобидные команды NOP. А вот с длиной строки нужно уложиться в определенные значения. Например, длина строки в 28 байт будет выглядеть как 28 и 0, что будет означать и INC E и NOP. Предпоследняя команда, в данном случае, увеличит регистр E на единицу.

Глава 5.

Создание и запись автозагрузочного блока кодов в машинный стек.

Краткое содержание: точки прерывания программы, регистр SP, подмена адреса возврата из машинного стека.

Давайте рассмотрим еще один метод автозагрузки блока кодов. На этот раз никаких скрытых BASIC команд не будет, и запускаться блок кодов будет только по `LOAD ""CODE` в нижнюю часть памяти, а именно в область машинного стека, который указывается в регистре SP. Об этом встречались упоминания в старых книжках, но детальное описание такой идеи, так нигде и не нашел. Попробуем разобраться.

Дело в том, что в этом стеке, помимо значений, компьютер еще хранит и адреса возврата из подпрограмм. Поэтому подменив значения в определенных ячейках, можно добиться перехода по нужному адресу. Для этого надо составить простую машинную программу, которая после окончания загрузки подменит значение в стеке, на адрес запуска своей программы. Но помимо своей программы, для успешного выхода в BASIC, необходимо восстановить значение нескольких ячеек стека, поврежденных в результате загрузки программы на них.

Первым делом «отловим» значение регистра SP, при выходе в BASIC, после успешного завершения `LOAD ""CODE`. Для начала запомним адрес дна стека, а также все значения, которые устанавливаются на чистом компьютере (после сброса). В Spectaculator откроем окно «*Debugger*», и посмотрим значение дна стека в переменной RAMTOP, которая расположена по адресу 23730 и 23731. Там находятся значения 87 и 255. Умножив значение ячейки 23731 на 256, и прибавив значение из «23730», получим адрес нижней границы стека. $255 \cdot 256 + 87 = 65367$. Посмотрев внимательно, увидим, что ниже того адреса идут рисунки букв, набираемых в режиме курсора `UDG`:

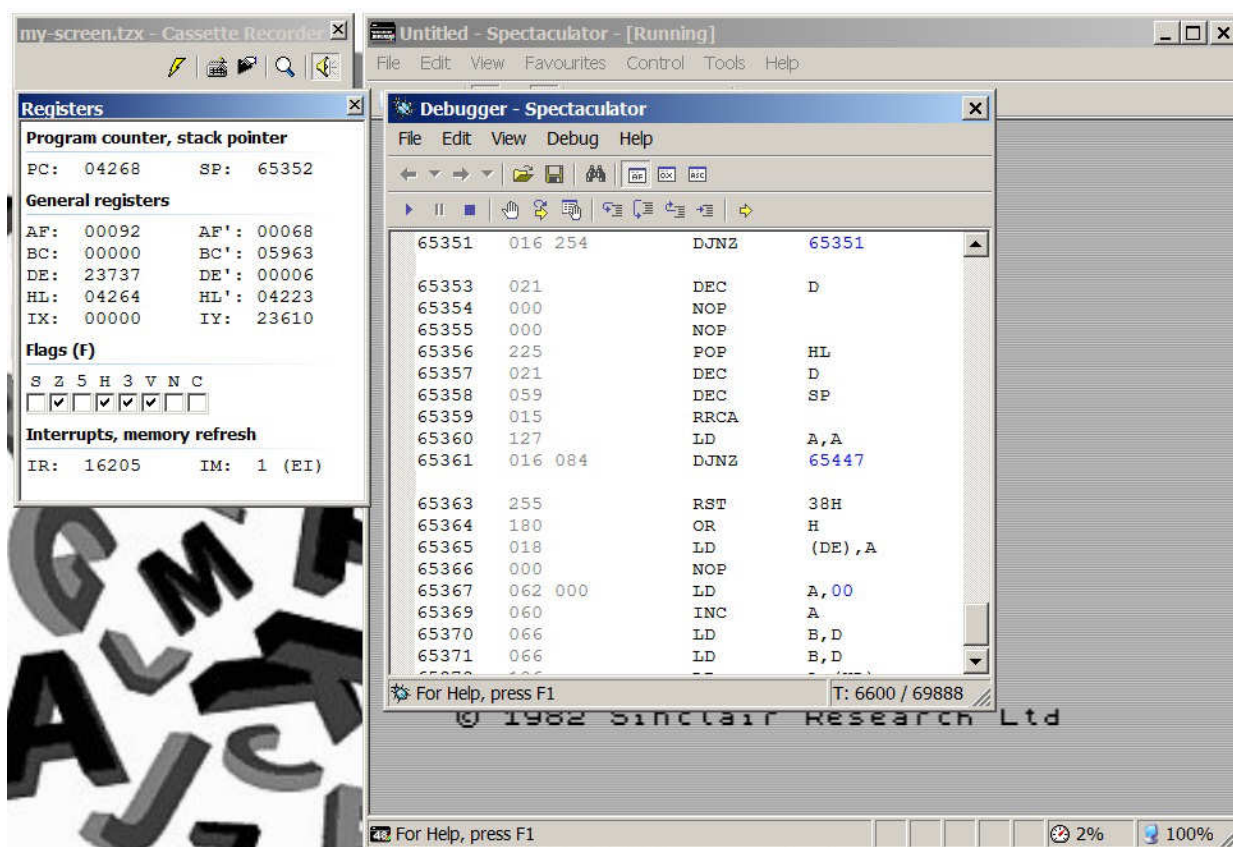


Рис. 2500. Состояние регистров, флагов и машинного стека «чистого» компьютера. Начало области UDG.

С нижней границей определились. Теперь запустим какой-нибудь блок кодов, и в процессе загрузки откроем «Debugger», изучив содержимое стека, и наставив точек прерывания (*Breakpoint*) по подозрительным адресам. В окне «Registers» будем наблюдать изменение значений регистра SP в разных ситуациях.

Вариантов тут немного, и в конце-концов придем к тому, что при выходе после `LOAD ""CODE` регистр SP опускается до отметки 65364, а следовательно, адрес перехода расположится в ячейках 65362 и 65363. Их значения в тот момент будут, соответственно, 118 и 27.

Несложно вычислить адрес, по которому происходит возврат после выполнения `LOAD ""CODE`. Адресом, по которому произойдет возврат в ПЗУ, будет 7030 ($27 \cdot 256 + 118$).

На всякий случай проверим это на практике. Начните загружать какой-нибудь блок данных, и откройте во время загрузки отладчик «Debugger». Видно, что значения стека начали сползать вниз, а по адресу 65362 и 65363, в отличие от «чистого» компьютера, стоят совершенно другие значения:

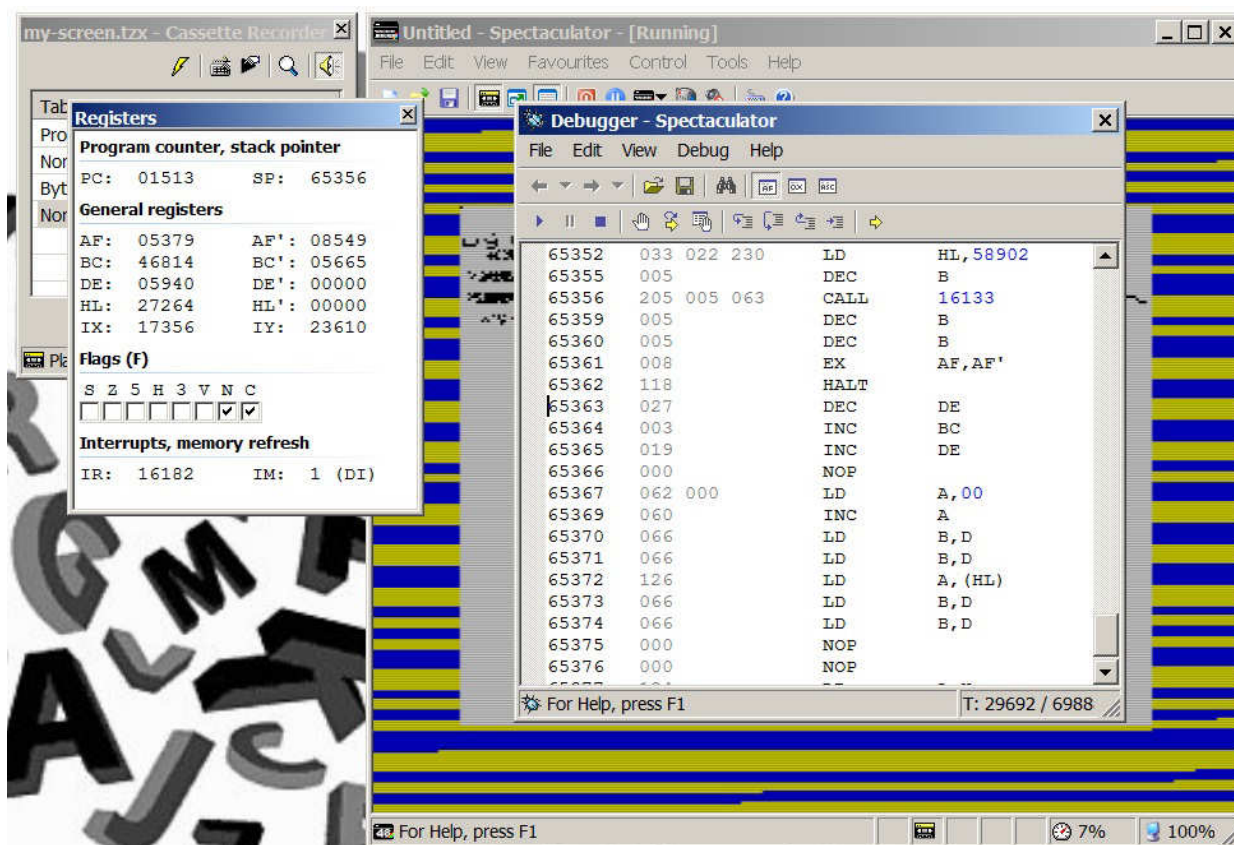


Рис. 2501. Значения ячеек стека во время загрузки блока данных на эмуляторе. Сползание вниз регистра SP.

Установим мышью курсор на адрес ПЗУ «7030», и поставим по этому адресу красную жирную точку (*Breakpoint*). Это можно сделать несколькими способами: Двойным нажатием левой кнопки мыши на указанном адресе, нажатием кнопочки с пиктограммой ладони или *CTRL+SPACE*. В результате на экране увидим следующее:

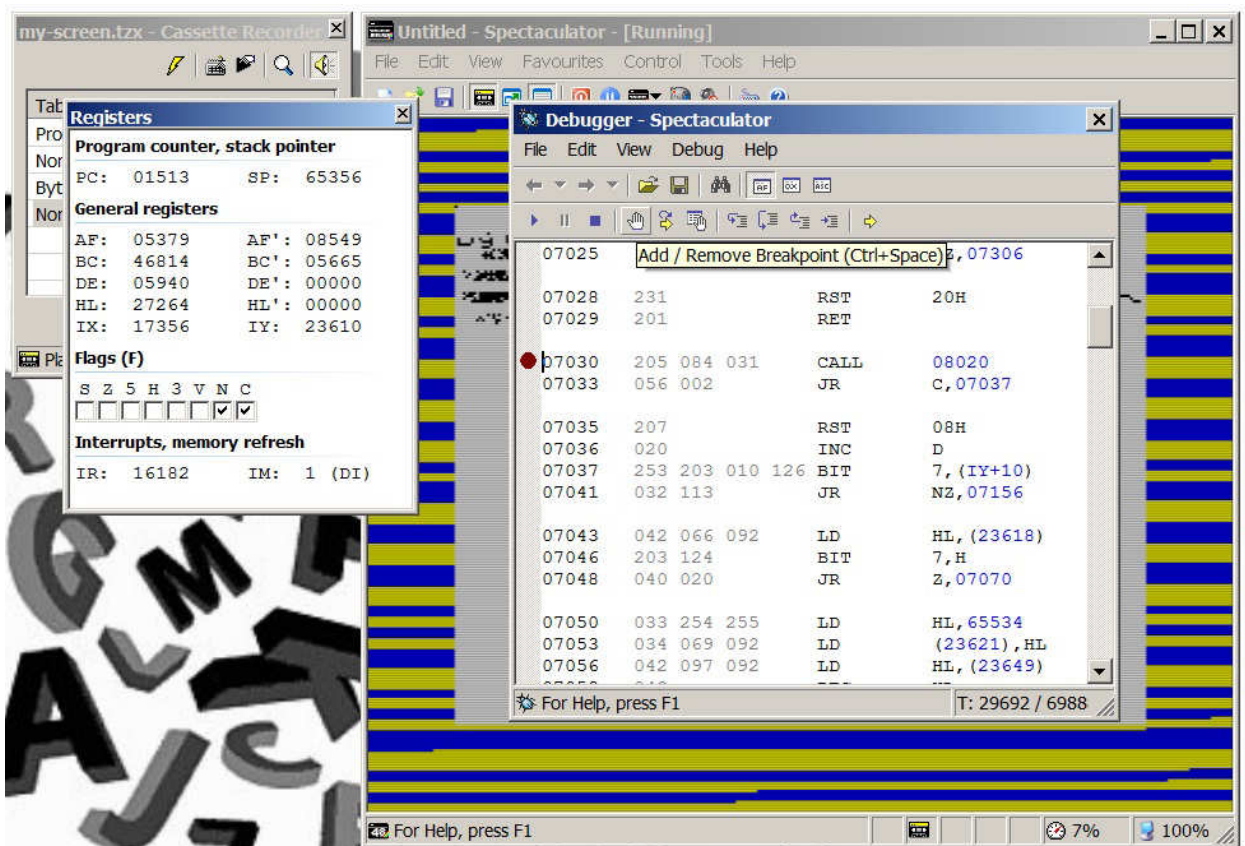


Рис. 2502. Установка точки прерывания в ПЗУ по адресу 7030 во время загрузки блока данных в эмулятор.

Эта точка будет означать, что при обработке команд с этого адреса, когда сюда попадет желтая стрелка, эмулятор остановится, и откроет окна в этом месте, со всеми текущими данными. Теперь закроем окно эмулятора, и компьютер продолжит загружать блок кодов. После загрузки эмулятор остановится, откроется «Debugger», а на красной точке появится желтый курсор, указывающий текущее выполнение машинной команды, который ожидает дальнейших действий. Значение стека SP в окошечке, при этом будет 65364. Таким образом, во время выполнения операций загрузки данных, он сполз на 10 байт вниз.

Это все что и требовалось узнать. Переписываем данные в стеке от адреса 65364 по 65367. Этими данными будут 3, 19, 0, 62. Снимаем точку прерывания повторным нажатием на нее, закрываем «Debugger» и перезагружаем эмулятор.

Теперь дело только за тестовой программой, которую придется начать с адреса 65362. В ячейки 65362 и 65363 придется поместить адрес перехода к нашей программе. Следом восстановление нескольких значений в стеке, и собственно, своя программа. В конце своей программы установим переход по адресу 7030 (118, 27). Сама тестовая программа будет располагаться с адреса 65368, поэтому первыми байтами, на подмену «118» и «27», будут «88» и «255».

Откроем EmulZWin. Первым делом в эмуляторе отнесем действующее значение стека, чтобы не произошел сброс во время написания программы. Для этого наберем и введем `CLEAR 39999`.

В окне ассемблера создадим простейшую программу и скомпилируем ее (убрав комментарии, стоящие через символ «;»):

```
ORG 65362
```

```
DEFB 88, 255 ; измененный адрес возврата, указывающий на тестовую программу
```

```
DEFB 3, 19, 0, 62 ; сохранение значений, которые там были изначально
```

```
LD A, 02 ; сама программа
CALL 5633
LD A, 90
RST 16
LD A, 88
RST 16
JP 7030 ; запоздалый переход к программе ПЗУ, которая должна была выполняться
изначально
```

Как видно, программа займет в памяти 20 байт. В нижней строке эмулятора наберем подготовительную программу:

```
SAVE «load-sp» CODE 65362,20
```

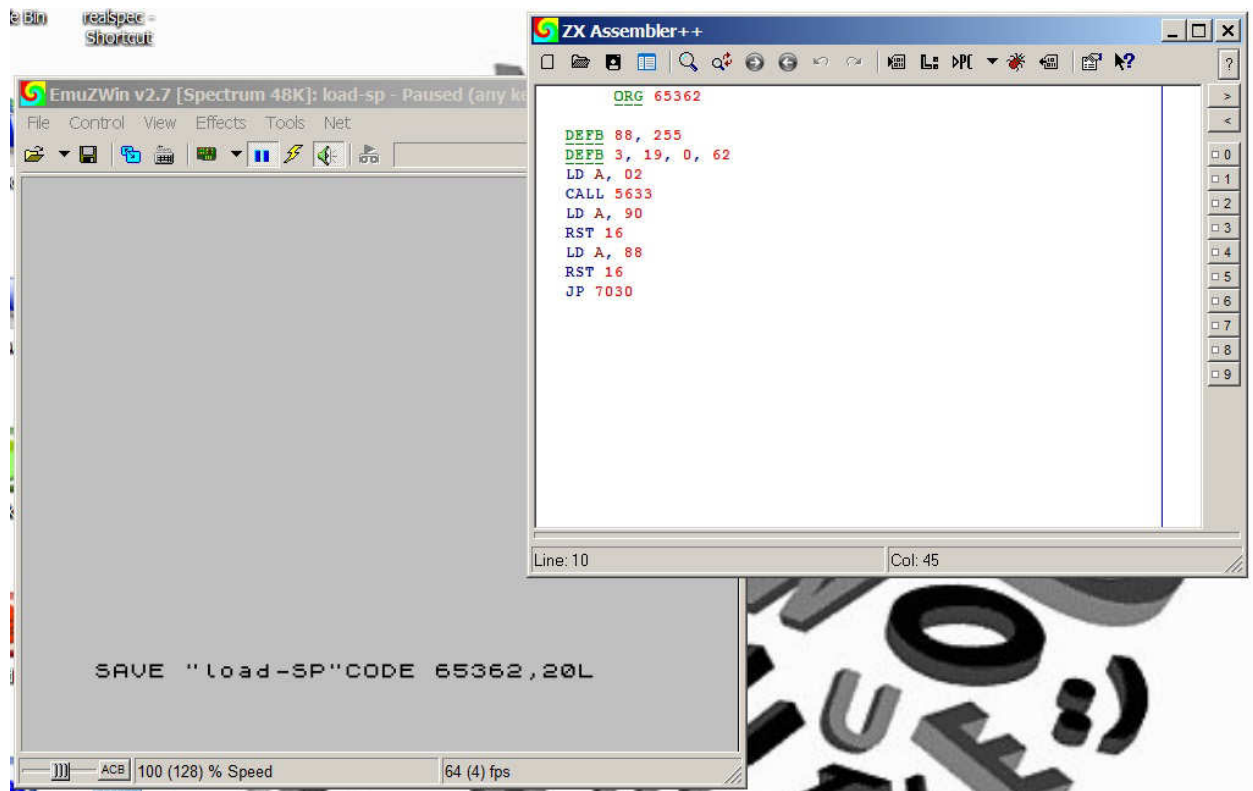




Рис. 2503. Подготовка "полуфабриката" программы к записи в *.tfx формате.

Сохраним программу под именем «load-sp.z80» и создадим удобным способом файл «load-sp.tfx».

Теперь запустим созданный файл на Spectaculator (если вы из него выходили) по LOAD "" CODE. После загрузки блока «Bytes : », программа автоматически стартует. Под заголовком высветятся буквы «ZX», а затем компьютер вернется к подпрограммам в ПЗУ и в нижней части экрана выдастся  OK ,  : 1:

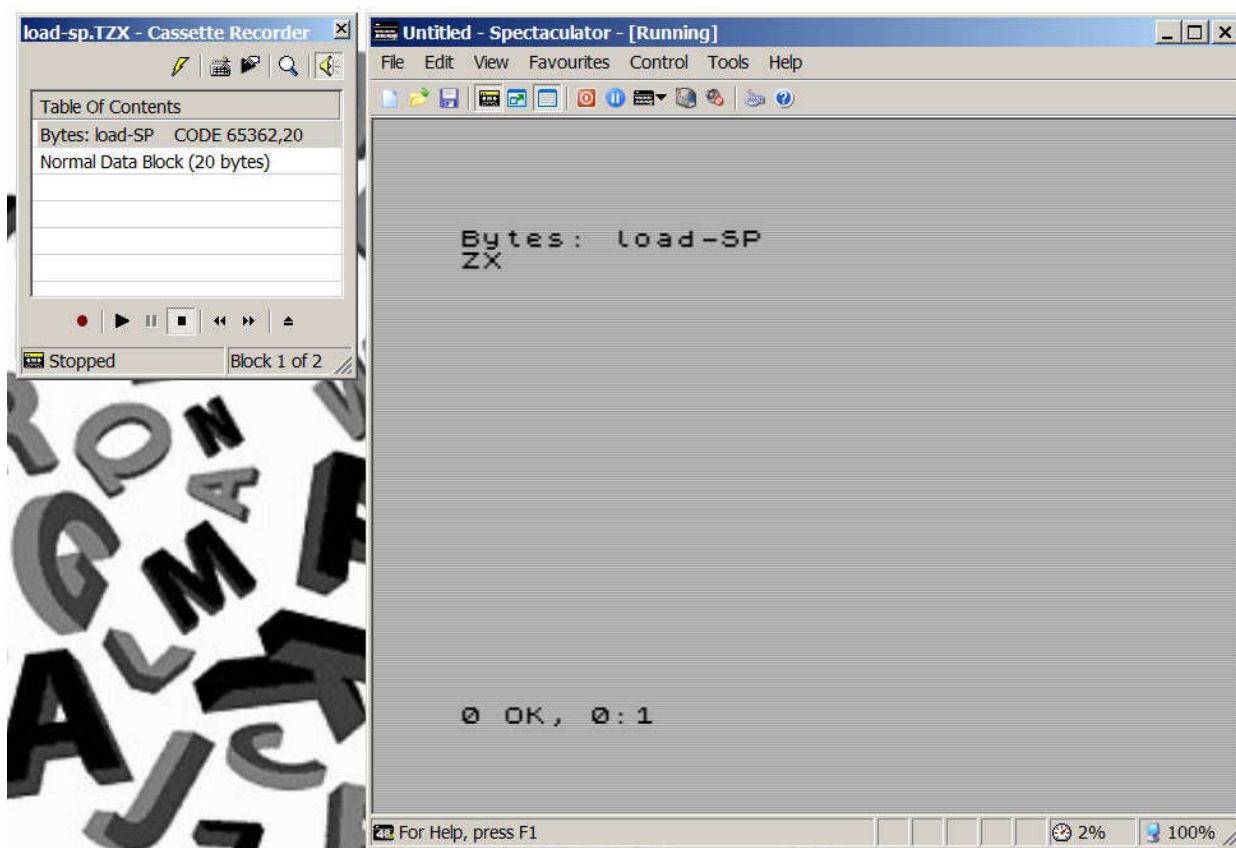


Рис. 2504. Проверочная загрузка и результат действия программы на эмуляторе Spectaculator.

В такой программе есть главное достоинство. Минимальное количество «паразитных байтов», которые следует включать в свою программу, для нормальной работоспособности системы после выполнения программы. При этом есть и недостатки – ограниченное пространство в нижней части памяти. Под программу можно использовать всего 168 байт нижних адресов. Так как программа пишется в область UDG, плюс ко всему прочему, повреждаются буквенные символы, которые доступны на клавиатуре в режиме курсора ⌂ .

Но можно попробовать модернизировать программу таким образом, чтобы избежать ограничения в размерах. Для этого данные нужно расположить так, чтобы они не начинались, а заканчивались в ячейках 65362 и 65363 адресом перехода. В этом случае ограничений длины не будет, и можно начинать загружать программу хоть с адреса 24800.

Для этого надо рассчитать свою программу так, чтобы она заканчивалась хотя-бы на 30-40 байт выше самых верхних значений стека (а лучше 50-60 если в программе часто используются PUSH и POP).

В этом случае в программу загрузки придется копировать значения верхней части стека, таким образом, чтобы их не повредить во время загрузки. В конце своей программы, для успешного выхода в BASIC и ПЗУ, также нужно поставить JP 7030.

Глава 6.

Загрузка машинной программы в область экрана.

Краткое содержание: машинная программа в экранной области, пиксельный мусор, переброска данных и запуск крограммы.

Наверняка кто-то замечал, что в некоторых программах, под конец загрузки (обычно взломанно-модифицированные программы) идет блок данных, который частично

искажает картинку «шумовым» эффектом. Обычно, таким образом, догружаются какие-то данные, а потом переносятся в определенную область памяти.

Проведем опыт и создадим классический 2-х блочный файл с BASIC-загрузчиком и такой «картинкой». Напишем простенькую, но в тоже время, довольно объемную программу, которая будет выводить текст на экран:

```
ORG 16384
LD HL,TEXT
LD DE, 30000
LD BC, 123
LDIR
JP 30000
TEXT: LD A,2
CALL 5633
LD DE,30015
LD BC,108
CALL 8252
RET
DEFB 22,9,2,16,1,18,1,"Prowerka zagruzki programmy"
DEFB 22,11,2,17,4,18,0,16,2,"w oblast ekrana, s perenosom"
DEFB 22,13,2,16,3,19,1,17,6,"i zapuskom ee s adresa 30000"
```

Откроем EmulZWin и наберем BASIC загрузчик, введя такую строку:

```
1 LOAD ""CODE: RANDOMIZE USR 16384
```

Откроем окно «ZX Assembler++», скопируем, или введем в окно ассемблера текст этой программы. Теперь перейдите в эмулятор, введите `LIST` с номером строки, большим чем имеется в программе. Пусть это будет `LIST 2`. Экран очистился, но таким образом можно избежать очистки экрана, после ввода строки, перед выдачей сообщения «`start tape, then press any key.`». Скомпилируем программу на ассемблере. На экране эмулятора ничего не произошло. Начните набирать подготовительную строку для записи:

```
SAVE "PROG16384" LINE 1: SAVE "prog16384"CODE
16384,137
```

Начав набирать команду `SAVE`, вверху экрана сразу появятся несколько черточек нашей программы в виде «помойки на экране»:

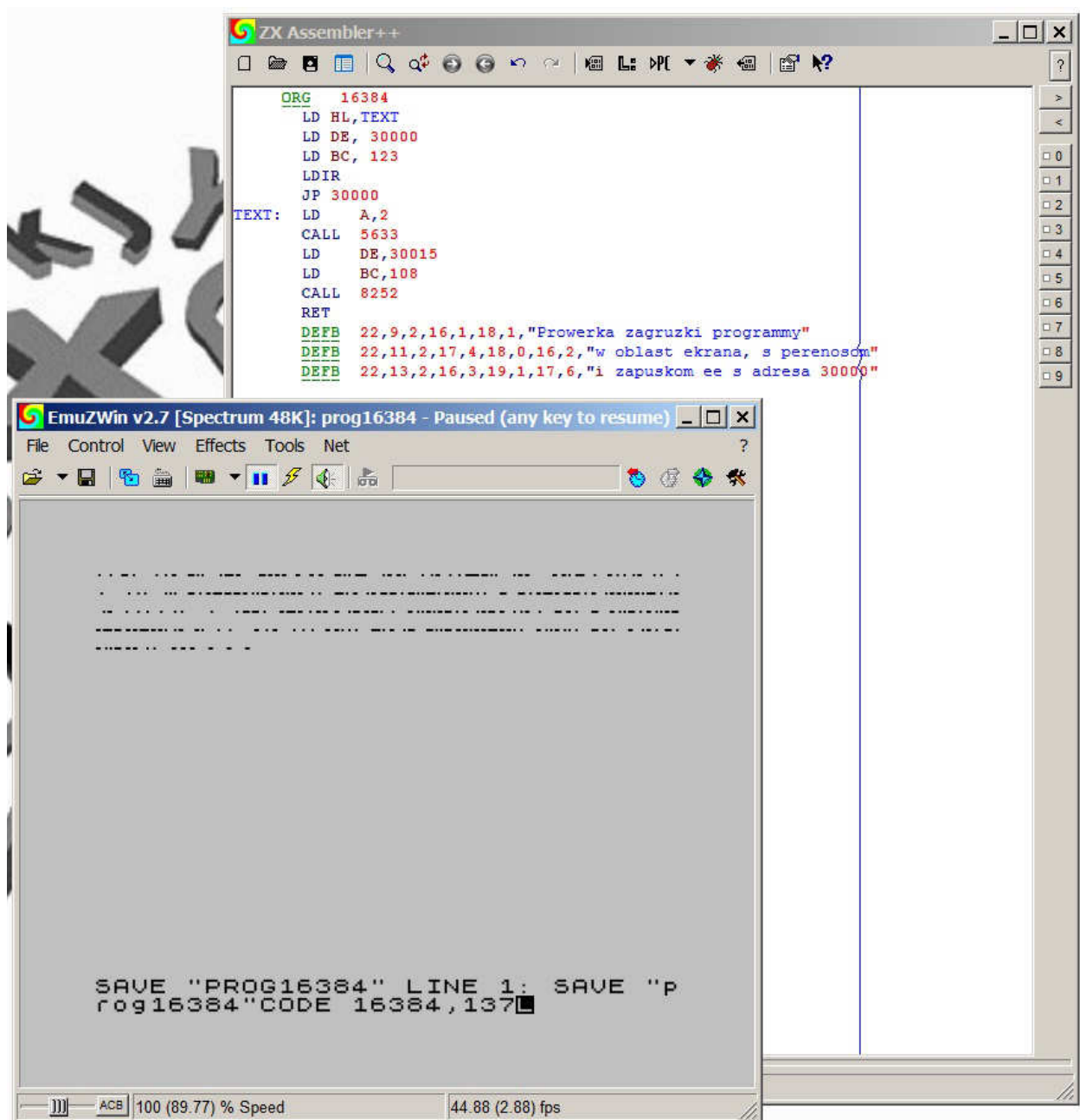


Рис. 2600. Отображение на экране программы в виде мусора. Подготовка программы к записи в .tzx.

Сохраните файл под именем «*prog16384.z80*». Создайте файл «*prog16384.tzx*», и попробуйте его запустить для проверки, набрав `LOAD ""ENTER`. После загрузки мы увидим, что программа выполнилась:

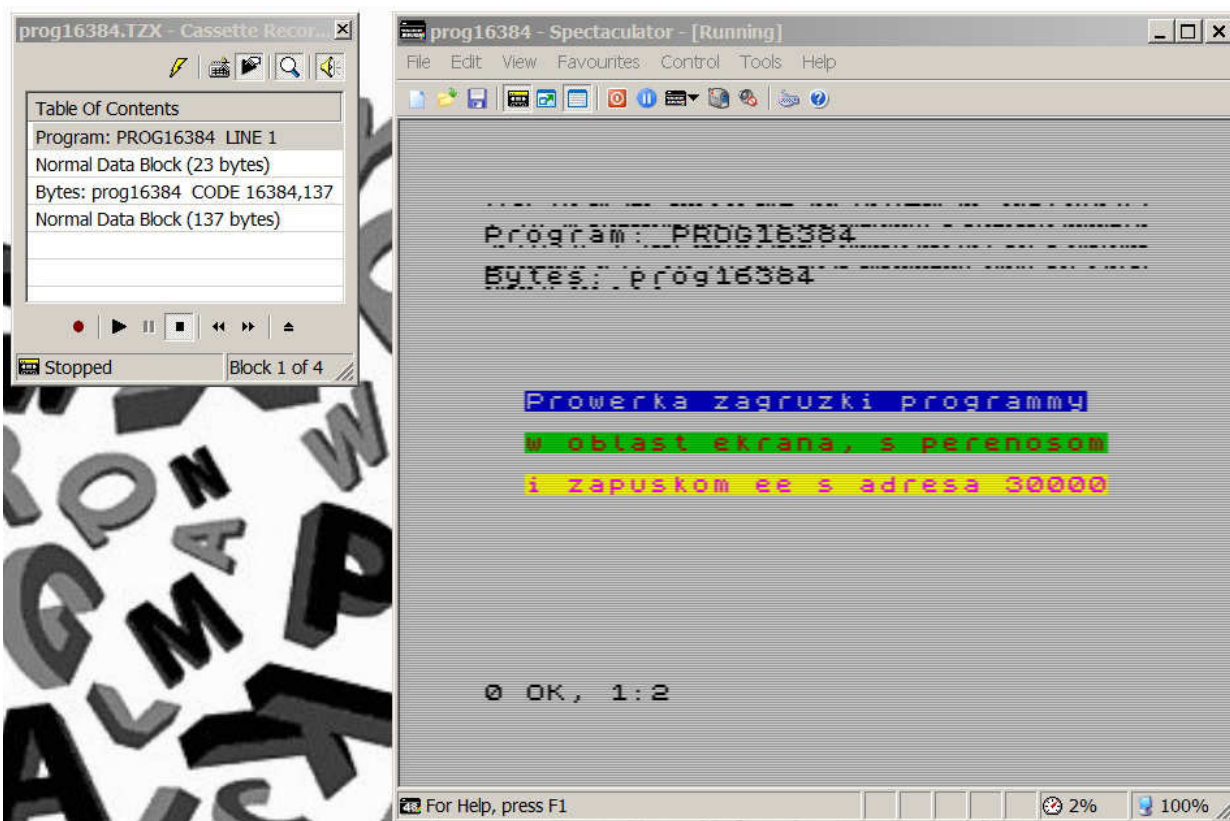


Рис. 2601. Полосы с мусором. Проверка работоспособности программы после загрузки.

Не стирая изображение «пиксельного мусора», можно снова запустить программу, набрав `RANDOMIZE USB 16384`. После того, как полосы с мусором будут стерты, или повреждены, запускать программу можно будет только по `RANDOMIZE USB 30000`.

Глава 7.

Загрузка и запуск блоков с переходом через предел 65535.

Краткое содержание: сохранение длинных блоков данных, «холостая» и «круговая» запись данных, каскадная программа, автозапуск I и II уровня.

Так как область ОЗУ спектрума очень мала, а чтобы создать приемлемую игру всегда требуется много места, то многие ресурсоемкие игры занимали практически всю память компьютера, порой подминая под себя каждый байт, включая системные переменные ОЗУ, и область экрана. Таким образом, длина программы могла достигать критического значения - 49152 байта. Учитывая то, что в ПЗУ реального спектрума было ничего не записать, а доходя до адреса 65535, компьютер снова переходит к считыванию данных по адресу 0, то блок можно удлинить до максимального значения – 65535 байт. Для 16-ти битного компьютера это будет предел длины.

Такая мера, в старые времена, одновременно являлась и хорошей защитой. Эту программу невозможно скопировать копировщиками, так как они тоже должны были размещаться в свободных адресах памяти, которых нет. А если еще этот загрузчик был модифицированным, и например, загрузку длинного блока вести в обратной последовательности от старших адресов к младшим, или нестандартным сигналом загрузки, то вскрыть ее было крайне сложно.

Интересным вариантом будет загрузка блока с переходом через предел «65535», в «0», проход через область экрана, переменных и далее. Получится «круговая запись», когда загрузка, пройдя через границу конца памяти 65535, перейдет к считыванию по адресу, 0 и начнет вхолостую накладывать данные на ПЗУ. Когда загрузка дойдет до

адреса 16384, данные снова начнут записываться в память, проявляясь на экране, пока не дойдут до предпоследнего байта, от которого начиналась загрузка. При таком способе, байты, наложенные на область ПЗУ, будут записаны в режиме холостого хода (при определенных настройках можно записать и в ПЗУ, но это пока не рассматривается), поэтому придется писать 16384 «паразитных байтов».

Давайте попробуем сделать блок данных, стартующий с последних адресов ОЗУ, проходящий максимальный адрес, и возвращающийся в ПЗУ. Чтобы не делать пустой бесполезный блок, попробуем создать сложный каскад, совместив две процедуры автостарта из предыдущих глав, посмотрев, какая из них будет иметь приоритет. Объединим несколько программ из этой части книги в этом опыте, заодно повторив все вышеизученное. Поставим цель, чтобы помимо загрузки информации, можно было осуществить корректный выход в BASIC с сообщением «OK . . .».

В теории, сначала должна запуситься программа, по переходу из машинного стека, затем выполнится введенная строка BASIC, из области **E_LINE**, и только потом произойдет запуск строк BASIC программы.

Общий алгоритм программы каскадного действия будет такой:

1. Загрузка через стек с автозапуском I уровня.
2. Выполнение программы автозапуска I уровня
3. Выполнение подгруженной «помойки» в область экрана.
4. Автозапуск II уровня вводимой BASIC-строки в **E_LINE**
5. Запуск из нее машинного кода
6. Запуск BASIC программы в строках, вызов 4-й машинной программы
7. Окончательный выход в BASIC систему.

Программа будет состоять из 4 автономных фрагментов данных, расположенных в разных местах памяти, которые будем записывать одним непрерывным блоком с переходом через предел 65535:

```
ORG 65362      ; Программа автозапуска I уровня
DEFB 88, 255
DEFB 3, 19, 0, 62
LD A, 02
CALL 5633
LD DE, TEXTSP
LD BC, 78
CALL 8252
JP 16384
```

TEXTSP: DEFB 22, 5, 0, 16, 2, "Avtozapusk programmy po perehoduiz mashinnogo steka - "

```
DEFB 22,6,22,16,0,17,7,18,1,"WYPOLNENO!"
```

```
ORG 16384      ; Программа "Помойка" в области экрана
LD HL,TEXT
LD DE, 30000
LD BC, 138
LDIR
JP 30000
TEXT: LD A,2
CALL 5633
LD DE, 30017
LD BC, 121
```

```
CALL 8252
JP 7030
DEFB 22,8,2,16,1,17,5,"Zagruzka programmy w oblast"
DEFB 22,9,1,17,4,18,0,16,2,"ekrana, s perenosom i zapuskom"
DEFB 22,10,1,16,3,19,1,17,6,"ee s adresa 30000 - "
DEFB 22,10,21,16,0,17,7,18,1,"WYPOLNENO!"
```

```
ORG 25000      ; 3-я машинная программа
LD  A,2
CALL 5633
LD  DE, TEXT3
LD  BC, 77
CALL 8252
RET
```

```
TEXT3: DEFB 22, 12, 0, 16, 5, 17, 2,"Zapusk mashinnogo koda iz komandnoy stroki
BASIC - "
```

```
DEFB 22, 13, 19, 16, 1, 17, 7, 18, 1,"WYPOLNENO!"
```

```
ORG 25500      ; 4-я машинная программа
LD  A,2
CALL 5633
LD  DE, TEXT4
LD  BC, 33
CALL 8252
RET
```

```
TEXT4: DEFB 22, 21, 1,"Prowerochnyi kaskad zakonchen!"
```

Теперь наберем BASIC программу, которая будет выполняться после всех процедур автозапуска:

```
1 PRINT AT 18,0;INK 3;"Zapusk stroki BASIC
programmy - "
2 PRINT INK 0; FLASH 1;"WYPOLNENO!"
3 RANDOMIZE USR 25500
```

В итоге, обе части программы, на экране, должны выглядеть вот так:

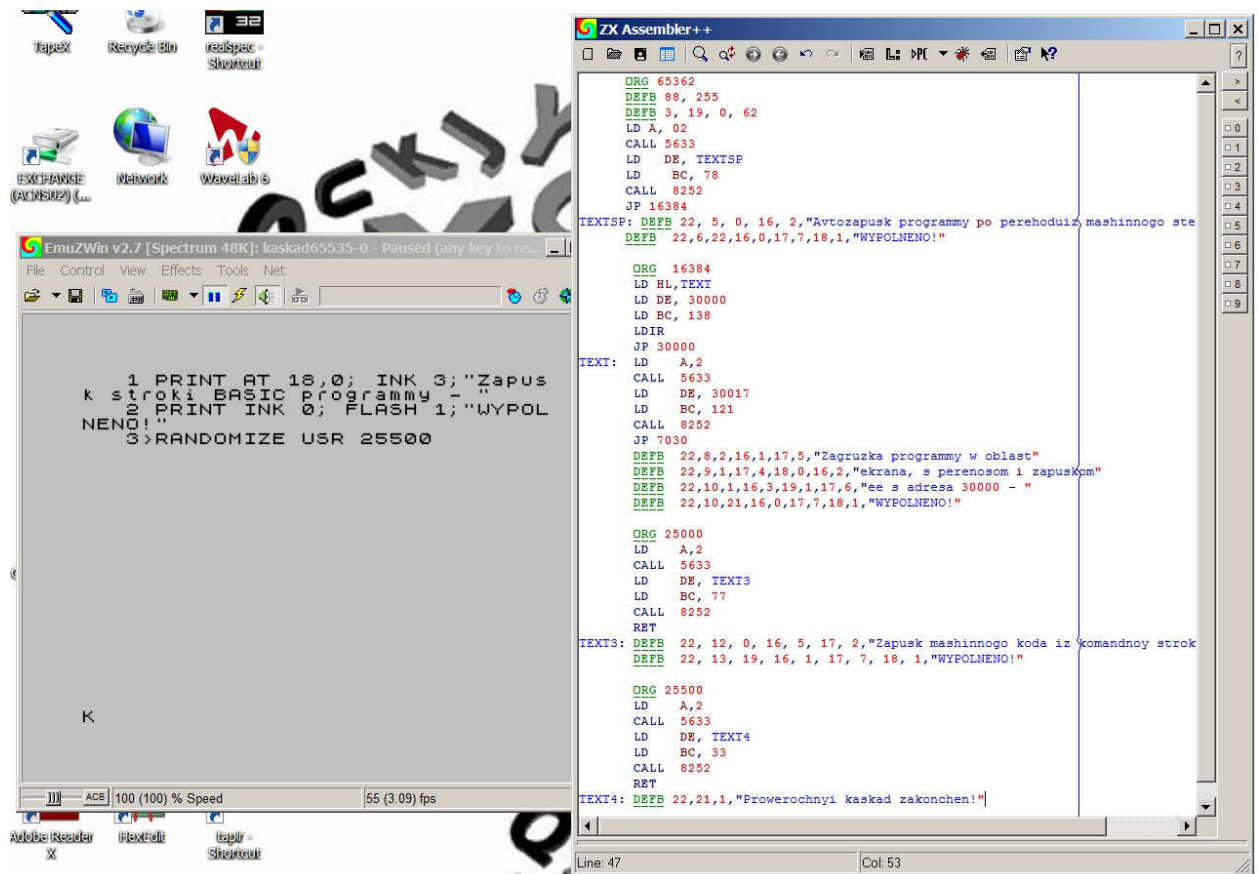


Рис. 2700. Программы на ассемблере и на BASIC.

Прежде чем приступить к написанию подготовительной строки, вычислим длину программы. Нижняя ее часть будет: $65536 - 65362 = 174$ байта. Верхняя часть будет длиной, на единицу больше байта последнего фрагмента программы. Это адрес 25548. Следовательно, полная длина будет: $25548 + 174 = 25722$

Введите команду **LIST 4** (или любую другую строку с номером больше чем 3), это убережет от очистки верха экрана во время выдачи сообщения на запись, а затем скомпилируйте программу в память эмулятора. Наберите подготовительную строку:

```

SAVE "65535 -> 0"CODE 65362,25722:
RANDOMIZE USR 25000: PRINT AT 15, 0;"Prowerka
zapuska wwodimoi stroki - ";FLASH
1;"WYPOLNENO": GO TO 1

```

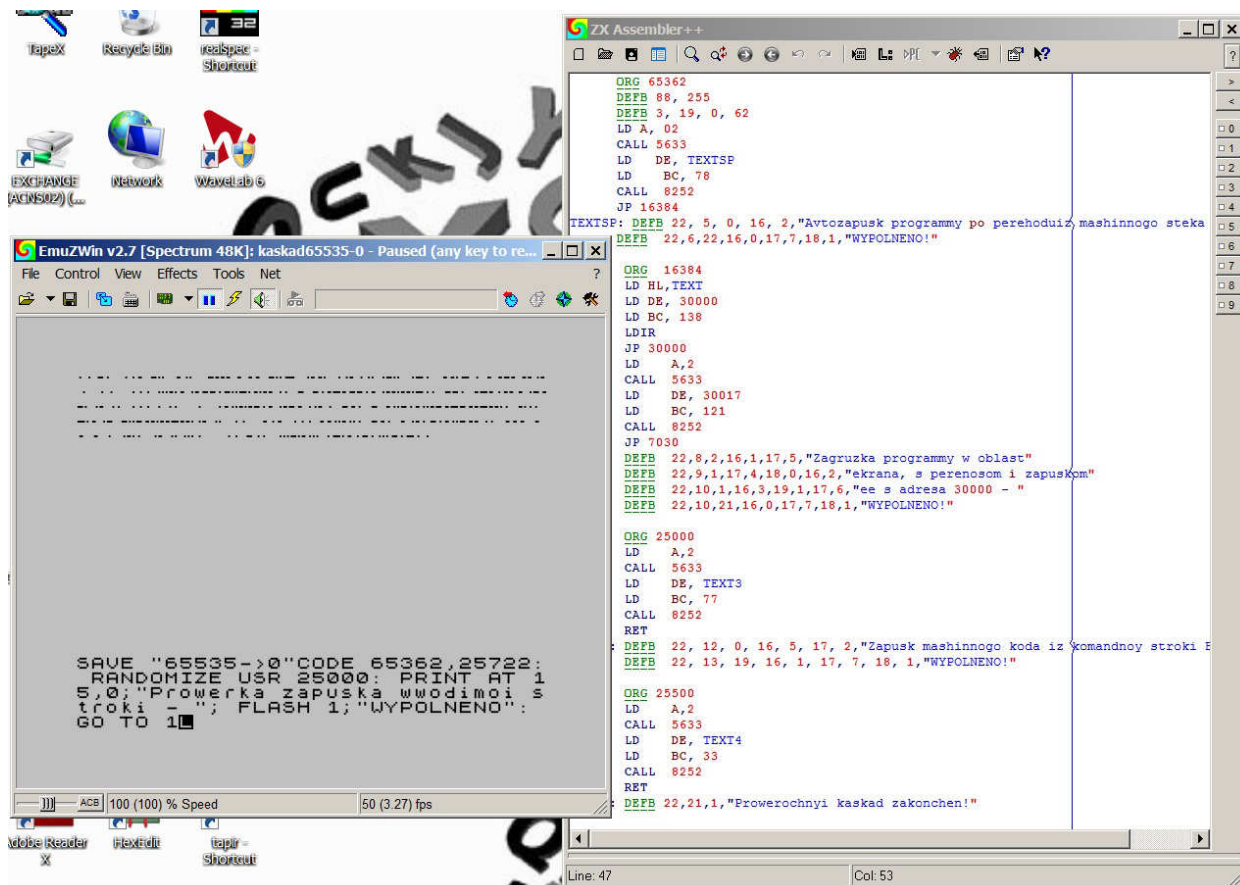


Рис. 2701. Полуфабрикат программы в EmuZwin перед записью в .z80 для создания .tzh

Сохраним как «*kaskad65535-0.z80*» и создадим одноименный файл «*kaskad65535-0.tzh*». Проверим в Spectasulator полученную программу. Во время перехода через максимальный адрес, и в ПЗУ, по рамке продолжают пробегать полосы, имитируя запись данных. В действительности, же никакой загрузки производиться не будет:

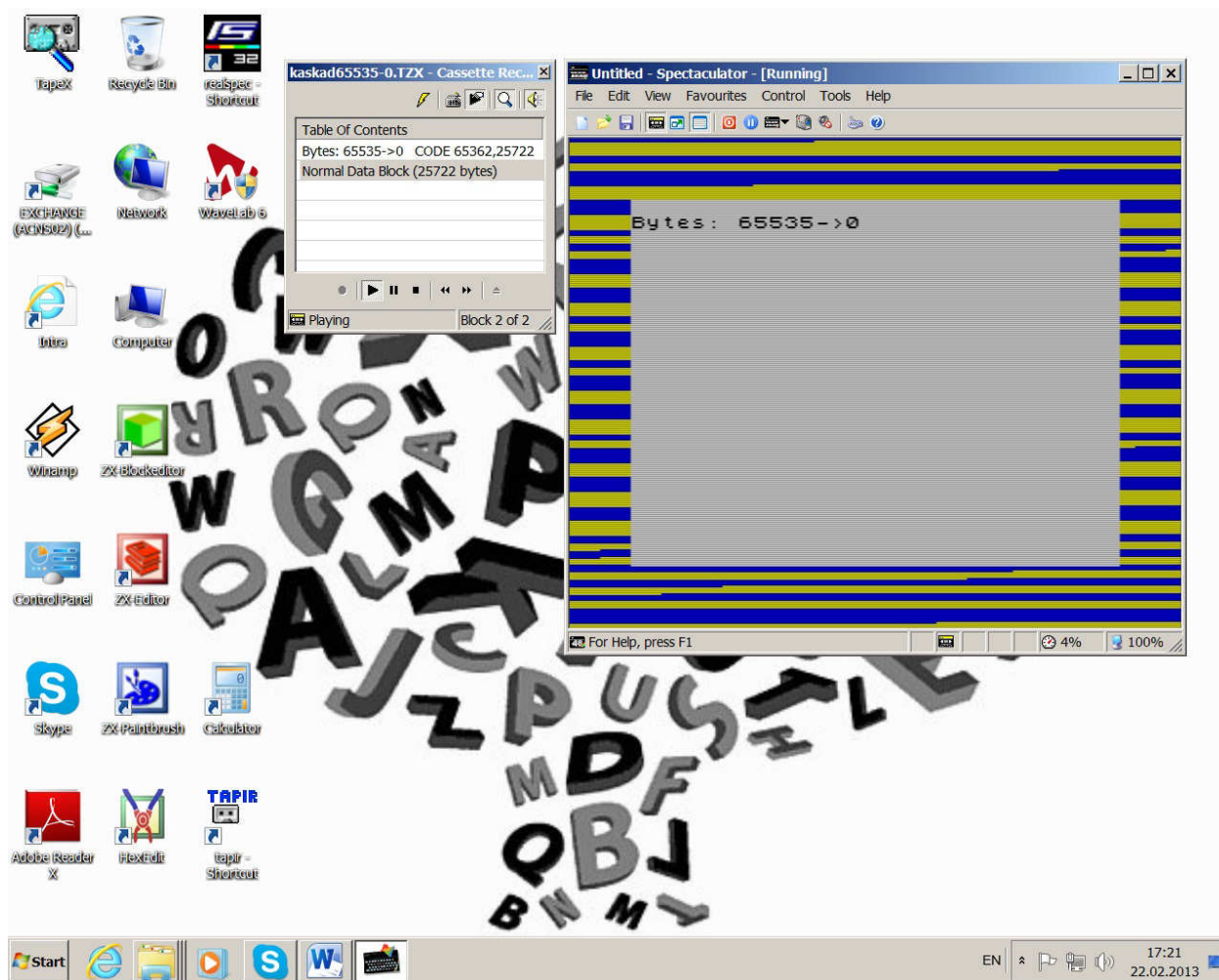


Рис. 2702. Программа в процессе «холостой загрузки» в ПЗУ на Spectaculator'e.

Во время перехода через порог ПЗУ/ОЗУ в область экрана загрузится программа в машинных кодах в виде мусора. Полностью загрузившись в эмулятор, первым делом программа перейдет по адресу в машинном стеке и выполнится строка:

«Avtozapusk programmy po perehodu iz mashinnogo steka - WYPOLNENO!»

Далее наша программа перейдет в область экрана, и перекопирует оттуда «пиксельный мусор» в адрес 30000. Запустится следующее контрольное сообщение:

«Zagruzka programmy w oblast ekrana, s perenosom i zapuskom ee s adresa 30000 - WYPOLNENO!»

Следом машинная программа переходит по адресу, которому должна была перейти при выходе из записи в BASIC. Тут интерпретатор BASIC подхватывает каскад и происходит автозапуск II уровня. Начинают выполняться остатки введенной строки в области E_LINE (После save она сохраняется в буфере) и программа снова уходит в машинный код, выдавая сообщение:

«Zapusk mashinnogo koda iz komandnoy stroki BASIC - WYPOLNENO!»

Возвращаясь из этой короткой программы по команде RET, продолжает выполняться вводимая строка, и на экране появляется следующее сообщение:

«Prowerka zapuska wwodimoi stroki – WYPOLNENO»

Далее, вводимая строка кончается, и по GO TO 1 компьютер переходит к выполнению обычной программы на BASIC со строками. 1 и 2 и выдается следующее сообщение:

«Zapusk stroki BASIC programmy – WYPOLNENO!»

В 3-й строке компьютер снова выполняет машинную программу и выдает заключительное сообщение:

«Prowerochnyi kaskad zakonchen!»

По возвращению в BASIC выдается сообщение OK, 3:1, означающее корректный выход в BASIC. Сложная каскадная программа, с двумя уровнями автозапуска, успешно выполнена. Это мы и видим на экране:

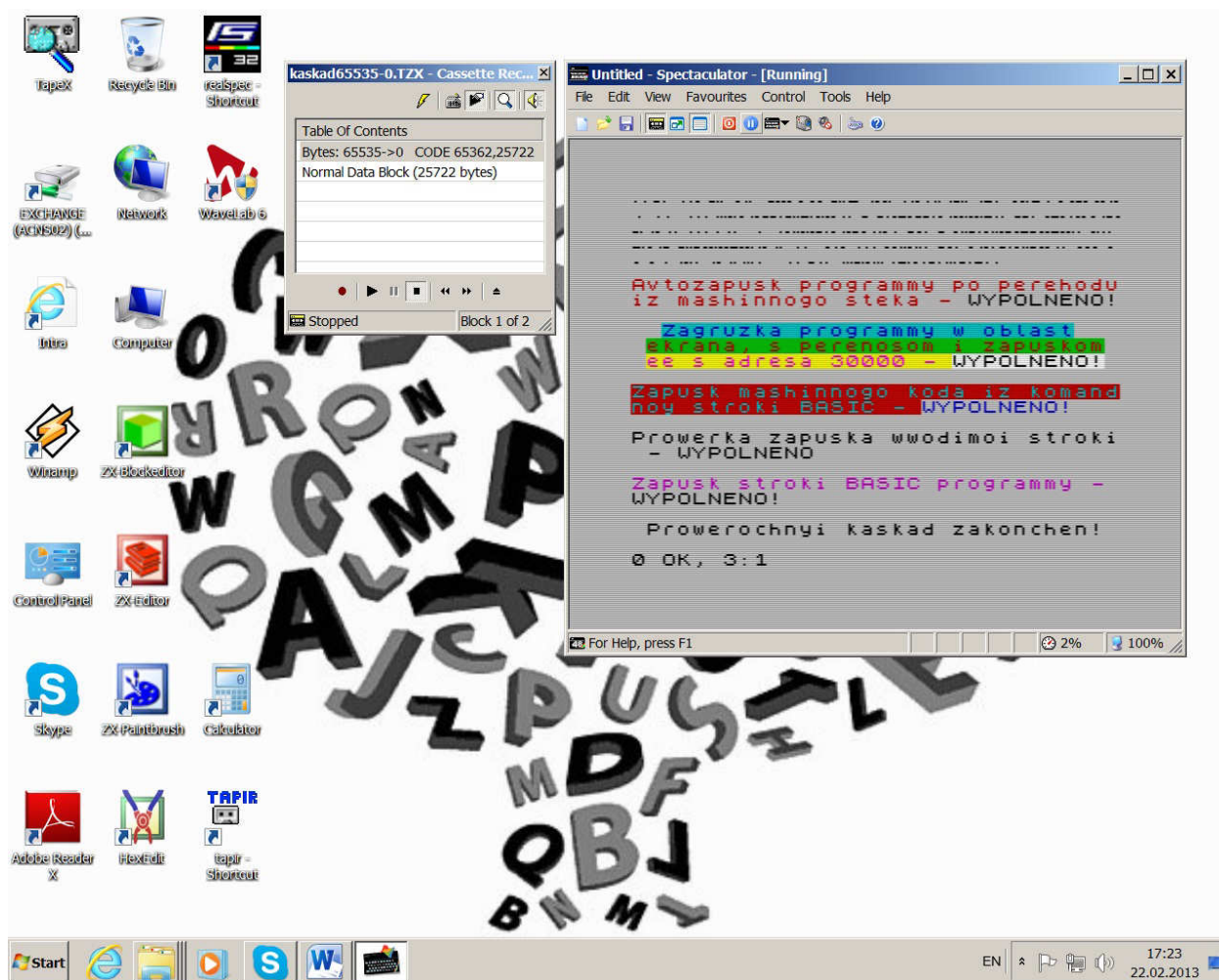


Рис. 2703. Успешное завершение многокаскадной программы с двумя уровнями автозапуска.

Можете поковыряться в строках программы, и в качестве проверки что-нибудь добавить или удалить.

Глава 8.

Простейшая загрузка файла без заголовка с измененными цветами полос.

Краткое содержание: подпрограммы считывания данных в ПЗУ, цветные полосы на рамке, запись файла без заголовка на эмуляторе.

В некоторых играх после основного блока, идет собственный загрузчик. Как правило, файлы загружаются без заголовка, а часто вместо полос обычного цвета, во время загрузки по рамке пробегают других цветов.

Стандартные полосы во время звука пилот-тона, красно голубые, одинаковой ширины, равномерно распределенные по всей рамке. Во время загрузки данных по рамке пробегают желто-синие полосочки. Широкие полосы соответствуют биту «1», узкие - биту «0». Бит состоит из пары смежных полосочек разного цвета. На картинке это хорошо видно:

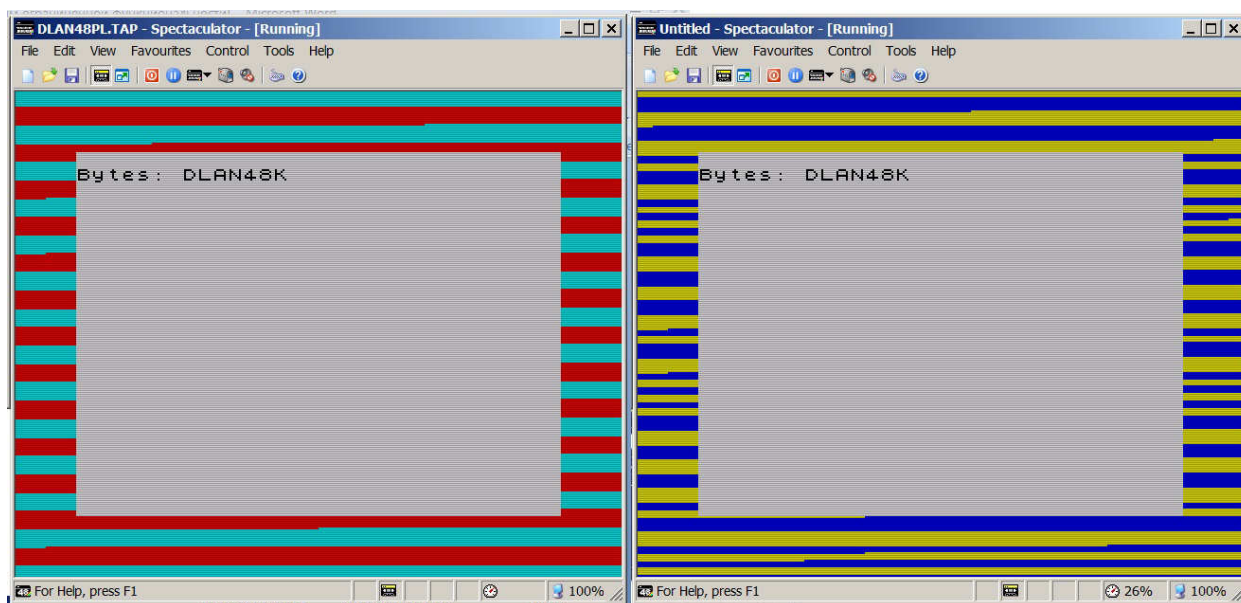


Рис. 2800. Стандартные цвета пилот-тона и данных процедуры загрузки из ПЗУ.

Давайте попробуем составить простейший загрузчик, с возможностью изменить цвет полосок при загрузке данных. Для этого вспомним загрузку и сохранение файлов из машинного кода по блокам. Перед вызовом программы (CALL 1218) следует указать длину блока, адрес загрузки, и тип передаваемых данных. Типовая программа сохранения блока данных выглядит следующим образом:

```
LD IX, стартовый адрес блока
LD DE, длина блока
LD A, тип блока (255 - данные, 0 - заголовок)
CALL 1218
RET
```

Загрузка выборочного блока данных вызывается подпрограммой из ПЗУ по адресу 1366. Перед вызовом CALL 1366 точно также следует указать длину блока, адрес загрузки и тип передаваемых данных:

LD IX, стартовый адрес блока
 LD DE, длина блока
 LD A, тип блока (255 - данные, 0 - заголовок)
 SCF
 CALL 1366
 RET

Наибольший интерес представляет загрузка, так как, изменив ее параметры, можно получить узор на рамке нестандартных цветов. Приступим к исследованию. Откроем эмулятор EmuZWin, и рассмотрим фрагмент программы из ПЗУ, которая начинается с адреса 1366:

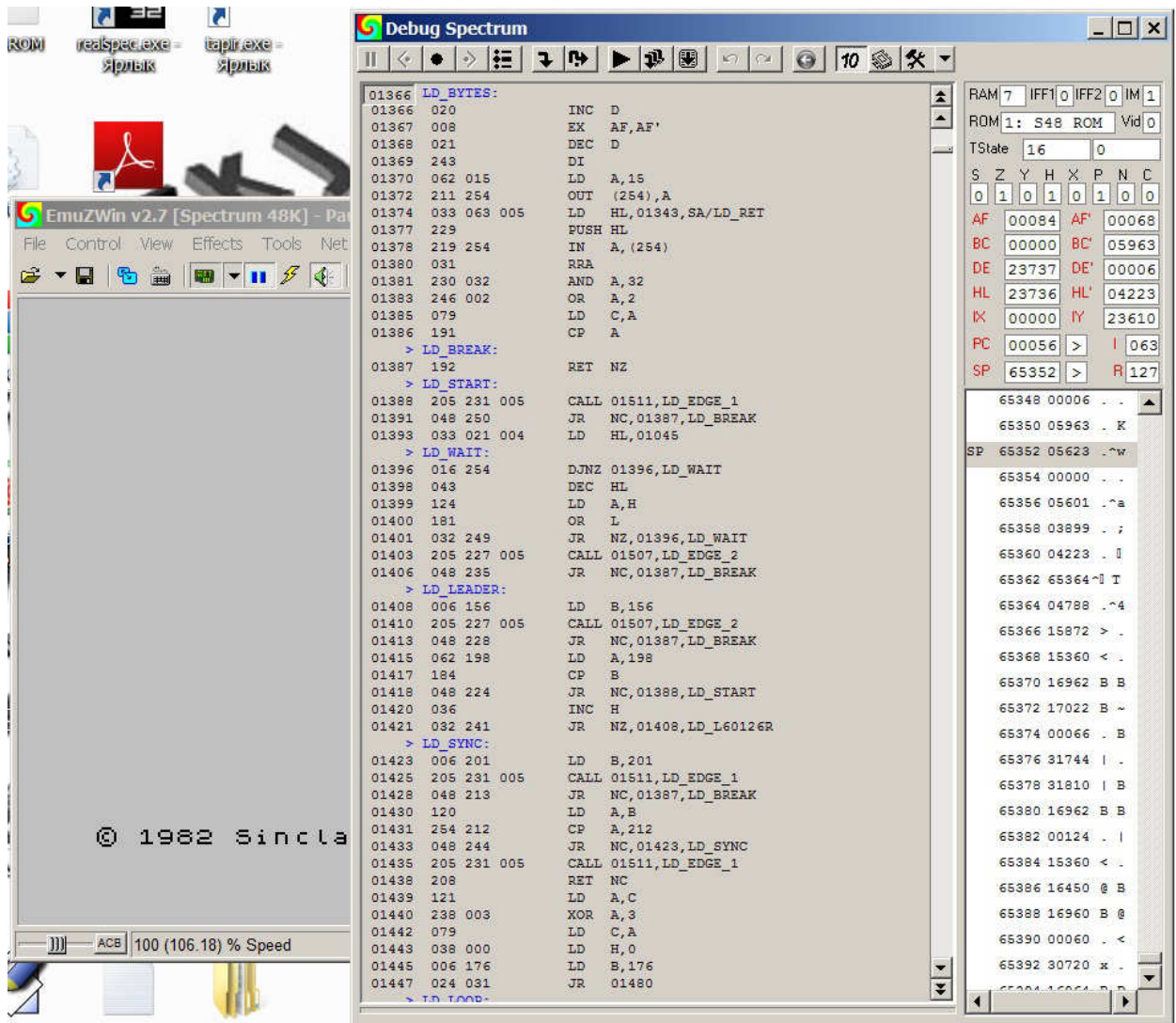


Рис. 2801. Отладчик Debug Spectrum. Начало программы загрузки данных в ПЗУ.

Даже если вы не очень хорошо разбираетесь в ассемблере, то посмотрев готовые промаркированные блоки программы загрузки и сохранения, можно в общих чертах разобраться, какой блок за что отвечает.

Дизассемблируем кусок программы ПЗУ с адреса 1366 по 1540. В окне «Select range to disassemble», помимо адресов, дополнительно поставьте галочки «Code in comments», «Jump/Call references» и «System labels». У вас сформируется большая часть подпрограммы загрузки из ПЗУ, с готовыми метками и подсказками, откуда и сколько раз вызываются те или иные фрагменты:

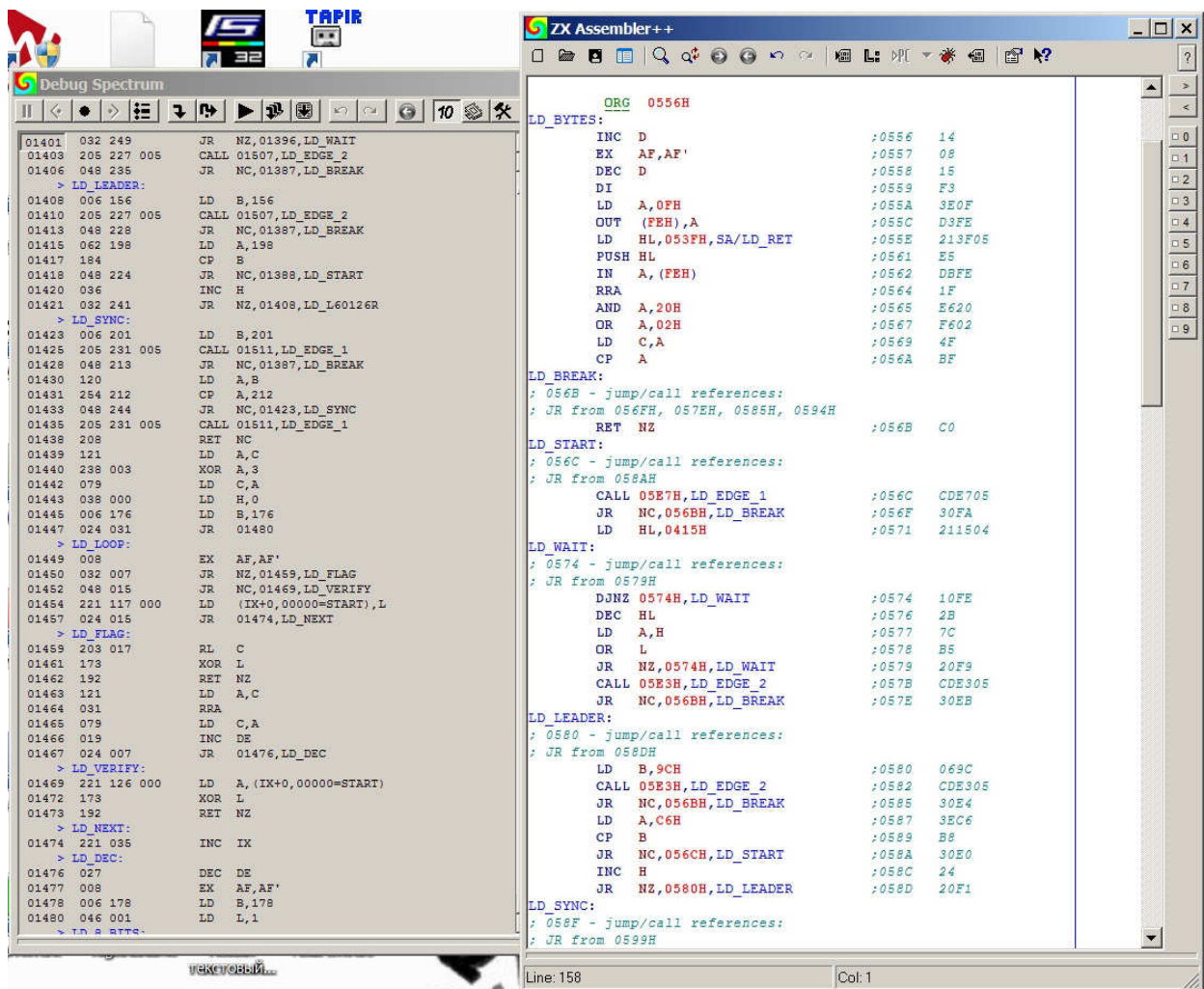


Рис. 2802. Процесс дизассемблирования фрагмента программы загрузки из ПЗУ.

Посмотрите на программу **LD_BYTES**. Оставим все остальное пока в покое, цель этой главы, создать программу минимум, чтобы как можно меньше байт процедур ПЗУ вытащить в свой загрузчик. Изучив ее внимательно, становится понятно, что начальный цвет рамки (после включения магнитофона вхолостую чередуется цвет рамки красный/голубой), устанавливается командой **OR 2**. Видим, что на подпрограмму **LD_BYTES** нет вызовов из других кусков программы. Значит, ее можно включить вместе с общей программой в ОЗУ, подменив фрагмент из ПЗУ. А играть с цветовой гаммой полосок будем путем изменения значений в команде **OR**. Вызов подпрограммы ПЗУ, вместо 1366 продолжим с точки **LD_BREAK** по адресу 1387.

Отсечем лишние блоки и подправим «выкушенный» из ПЗУ текстовый фрагмент программы, изменив точку входа. Можно сохранить его в блокноте. Припишите сверху к нему начальные данные, например, блок. Шаблон готовой программы будет выглядеть так:

```

ORG 30000
; основные данные
LD IX, 35000
LD DE, 3000
LD A, 255
SCF
; продолжение программы LD_BYTES
INC D
EX AF,AF'

```

```

DEC D
DI
LD A,15
OUT (254),A
LD HL,1343
PUSH HL
IN A,(254)
RRA
AND A,32
OR A,2 ; начальный цвет рамки пилот-тона красный
LD C,A
CP A
CALL 1387
RET

```

Программа занимает 35 байт памяти. Помимо нее придется еще создавать программу записи одиночного блока данных, которая и будет загружаться полосками нестандартного цвета. Поместим ее с адреса 40000, и скомпилируем вместе с загрузкой. Пусть это будет фрагмент ПЗУ. В данном случае содержание данных не имеет значения, так как будем акцентировать внимание на самом процессе загрузки:

```

ORG 40000

LD IX, 0
LD DE, 3000
LD A, 255
CALL 1218
RET

```

Поменяем в своей программе базовый OR 2, на OR 1, нажмем RESET на эмуляторе, и скомпилируем в него программу на ассемблере.

Теперь составим программу-загрузчик на BASIC, для считывания файлов без заголовка. Введите строку:

```
1 LOAD ""CODE: RANDOMIZE USR 30000
```

Программа готова. Теперь наберем в нижней части экрана строку, для подготовки полуфабриката:

```
SAVE "POLOS" LINE 1: SAVE "polos"CODE
30000,35 :RANDOMIZE USR 40000
```

Заметим, что RANDOMIZE USR 40000 в эмуляторе будет производить запись последнего блока без заголовка. Выглядеть программа будет так:

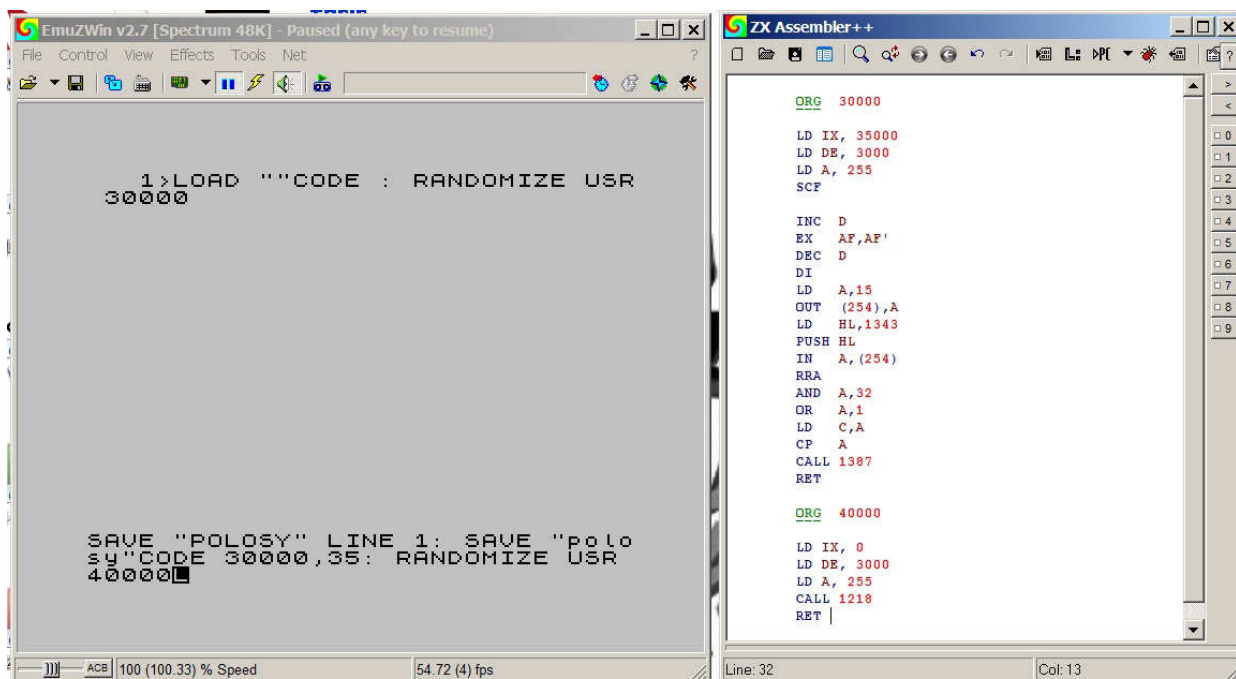


Рис. 2803. Подготовка программы с блоком кодов без заголовка и нестандартным цветом полос.

Сохраним файл под именем «*polosy.z80*». В Spectaculator или Realspectrum создадим «*polosy.tzx*». Заметим, что оба эмулятора поддерживали процедуру нестандартной записи, и корректно вписали блок кодов без заголовка в файл *.tzx по команде **RANDOMIZE USR 40000**.

Теперь проверим, что у нас получилось, понаблюдая за загрузкой последнего блока на Spectaculator:

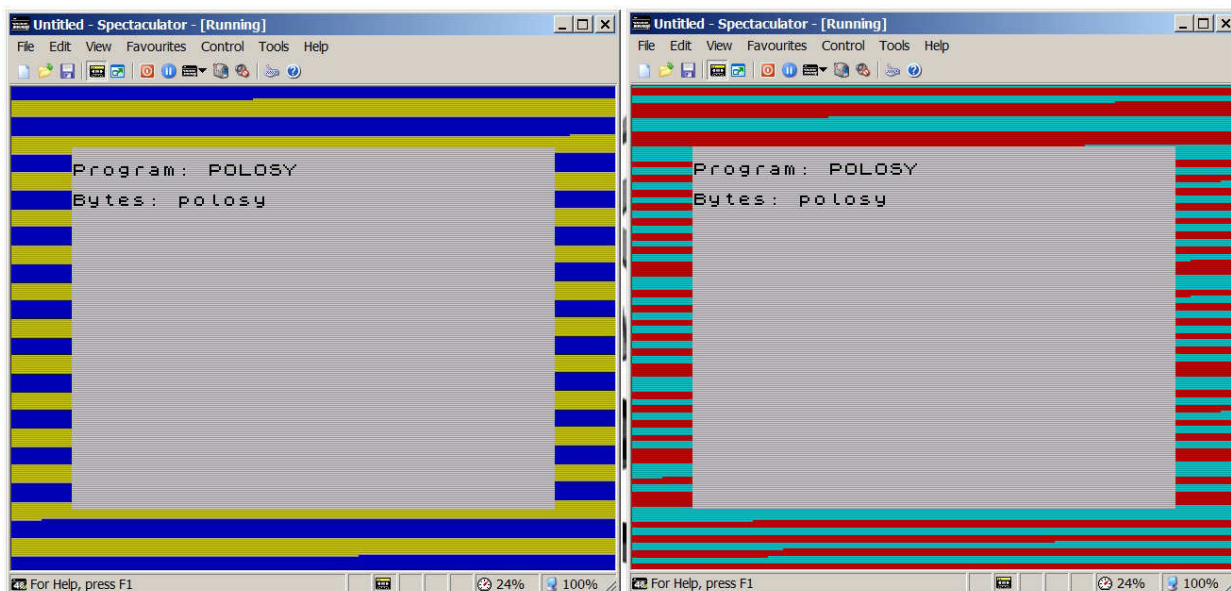


Рис. 2804. Процесс загрузки программы. Изменение цвета полос пилот-тона и данных.

У нас все поменялось. Теперь сигнал пилот-тона отображается в виде желто-синих полос, а сигнал данных - красно-голубых. Таким образом смена OR 2 на OR 1 дает стартовый цвет пилот-тона синий, при этом цветовой набор останется прежним. Попробуйте в «*Debugger'e*» вводить разные значения для OR от 0 до 7, запуская последний блок кода без заголовка, и наблюдайте какие комбинации цветов будут использоваться.

Результаты некоторых опытов со значениями даны ниже:

Значение команды OR	Цвет полосок пилот-тона	Цвет полосок данных
0	Черно-Белые	Зелено-Фиолетовые
1	Синие-Желтые	Голубые с Красным
2 (классический)	Красно-Голубые	Желто-Синие
3	Фиолетово-Зеленые	Бело-Черные
4	Зеленые-Фиолетовые	Черно-Белые
5	Голубые с Красным	Сине-Желтые
6	Желто-Синие	Красно-Голубые
7	Бело-Черные	Фиолетово-Зеленые

Как видно из таблицы, при такой модификации, возможно получить только 4 цветовых набора, потому что черно-белый и бело-черный при загрузке будут выглядеть одинаково. Отличие будет только в стартовой рамке.

Глава 9.

Запись картинки с нижними строками и длинным заголовком с управляющими кодами.

Краткое содержание: запись блока картинки с нижними строками, расчет фрагментов области экрана, запись заголовка, регулировка зазора между блоками, управляющие символы в имени заголовка.

Многие сталкивались, когда после создания картинки и попытки записи на магнитофон, обычными средствами BASIC в нижней строке выскакивает надпись "start tape then press any key", которая подлым образом стирает кусок изображения с нижних строк экрана. Команда LIST в этом случае помогает сохранить изображение в строках основной части экрана.

Но, тем не менее, при загрузке многих игр, картинка загружается с нижними строками. Люди приспособились, и нашли много вариантов устранить это недоразумение. На настоящем спектре сделать это обычной командой SAVE было невозможно. С эмулятором такая возможность появилась. Нужно просто в процессе записи, сразу после нажатия клавиши (start tape, then press any key), в ожидании пилот-тона, быстро нажать паузу эмулятора и скомпилировать картинку в область ОЗУ. После этого сохранить слепок памяти *.z80 и преобразовать в *.tzh в Realspectrum'e. В этом случае теряется часть сигнала пилот-тона заголовка.

Но есть и другой, более интересный вариант. Давайте попробуем по отдельности создать блок заголовка, и блок картинки с нижними строками. Мало того имя заголовка будет цветное и состоять более чем из 10 букв подряд.

Создайте тестовую картинку с нижними строками, например запуском BASIC программы:

```
1 PRINT #0; INK 2; PAPER 6;"   Prowerka
zapisi i zagruzki   kartinki w nizhnie stroki
ekrana"
2 PAUSE 0
```

Запустите программу, и команда PAUSE 0 заморозит картинку в нижних строках. В это время перейдите в окно ассемблера. Чтобы не вынимать гору пустых байтов, дизассемблируем только нижнюю треть экрана, и целиком область цветных атрибутов. Вся область экрана занимает 6144 байта. Треть экрана будет 2048 байт, следовательно, начало нижней трети будет 16384+2048*2=20480. Полная длина фрагмента данных будет

2048+768=2816 байт. Дизассемблируйте этот кусок картинки с начальным адресом 20480 и длиной 2816 байт в виде байтов DEFB:

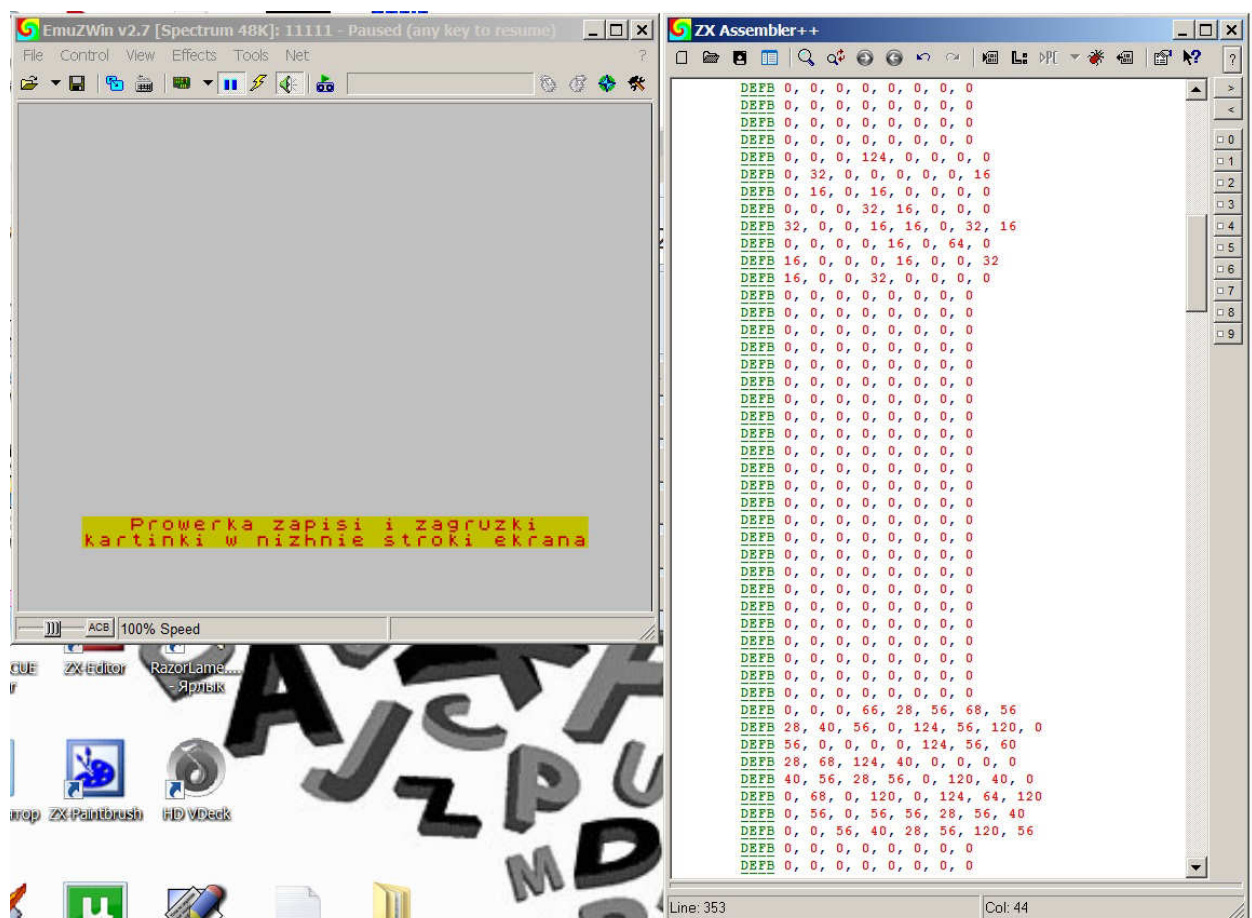


Рис. 2900. Дизассемблирование фрагмента картинки в нижних строках.

Теперь вспомним, из каких основных частей состоит заголовок блока «Bytes : » его структуру, и длину:

Тип файла (3-Bytes) (1 байт)

Заголовок. Неиспользуемые символы заполняются пробелами до 10 байт.

Длина блока данных (2 байта)

Стартовый адрес (2 байта)

Не используются для «Bytes:» (2 байта)

Итого длина заголовка 17 байт. Самое интересное, что в имени, кроме букв, можно помещать управляющие символы (AT с координатами, TAB), включая цветные, и целые команды BASIC. Так как команда BASIC состоит из набора символов, то написав код команды, в заголовке выскочит целое слово. Очевидно, что самым длинным заголовком может стать 10 команд RANDOMIZE подряд. Давайте создадим цветной заголовок «Bytes : », состоящий из команд и букв, назвав его: «SCREEN\$ from RANDOMIZE»

Напишем подготовительную программу для последовательной записи заголовка и блока данных в адрес 30000 и 30020. Данные для заголовка (имя, длину и адрес) поместим в 22980. Получится такая программа:

```
ORG 30000
```

```
LD IX, 29980
```

```
LD DE, 17
```



```
LD A, 0
CALL 1218
RET
```

```
ORG 30020
```

```
LD IX, 20480
LD DE, 2816
LD A, 255
CALL 1218
RET
```

```
ORG 29980
DEFB 3, 16, 2, 170, 17, 1, 102, 114, 111, 109, 249
DEFB 0, 11
DEFB 0, 80
DEFB 0, 0
```

Вставим три куска программы перед фрагментом картинки, а эмулятор очистим от программы, которая больше не нужна, нажав Reset:

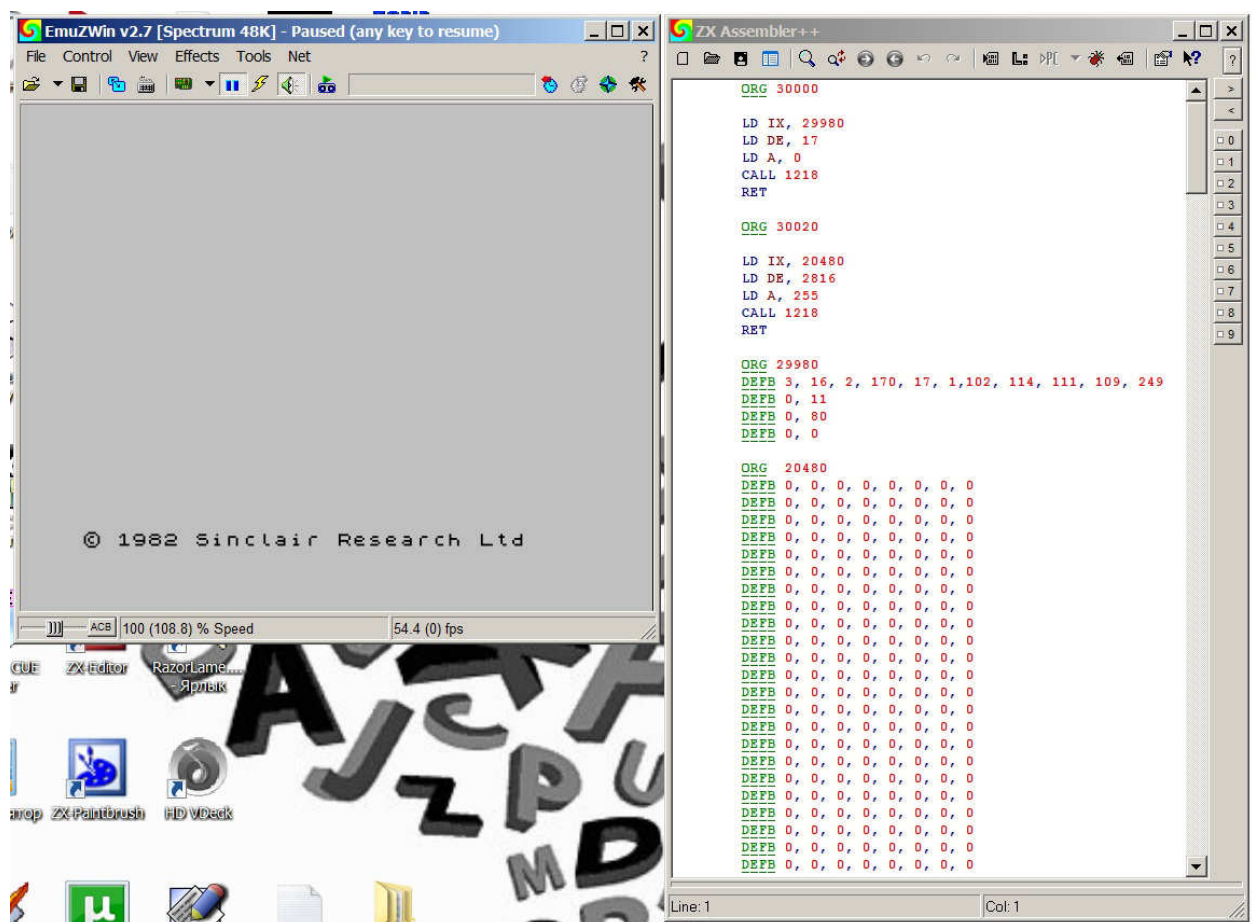


Рис. 2901. Очистка памяти компьютера и подготовка к компиляции.

В нижней строке наберем подготовительную программу:

```
PAUSE 0: RANDOMIZE USR 30000: PAUSE 50:
RANDOMIZE USR 30020
```

Команда `PAUSE 50` нужна для создания зазора между блоком заголовка и данными.

Введем строку, и она зависнет в ожидании на `PAUSE 0`. Ничего дальше не трогая, скомпилируем полученную программу с картинкой и блоками, сохранив под именем «`randscreen.z80`»:

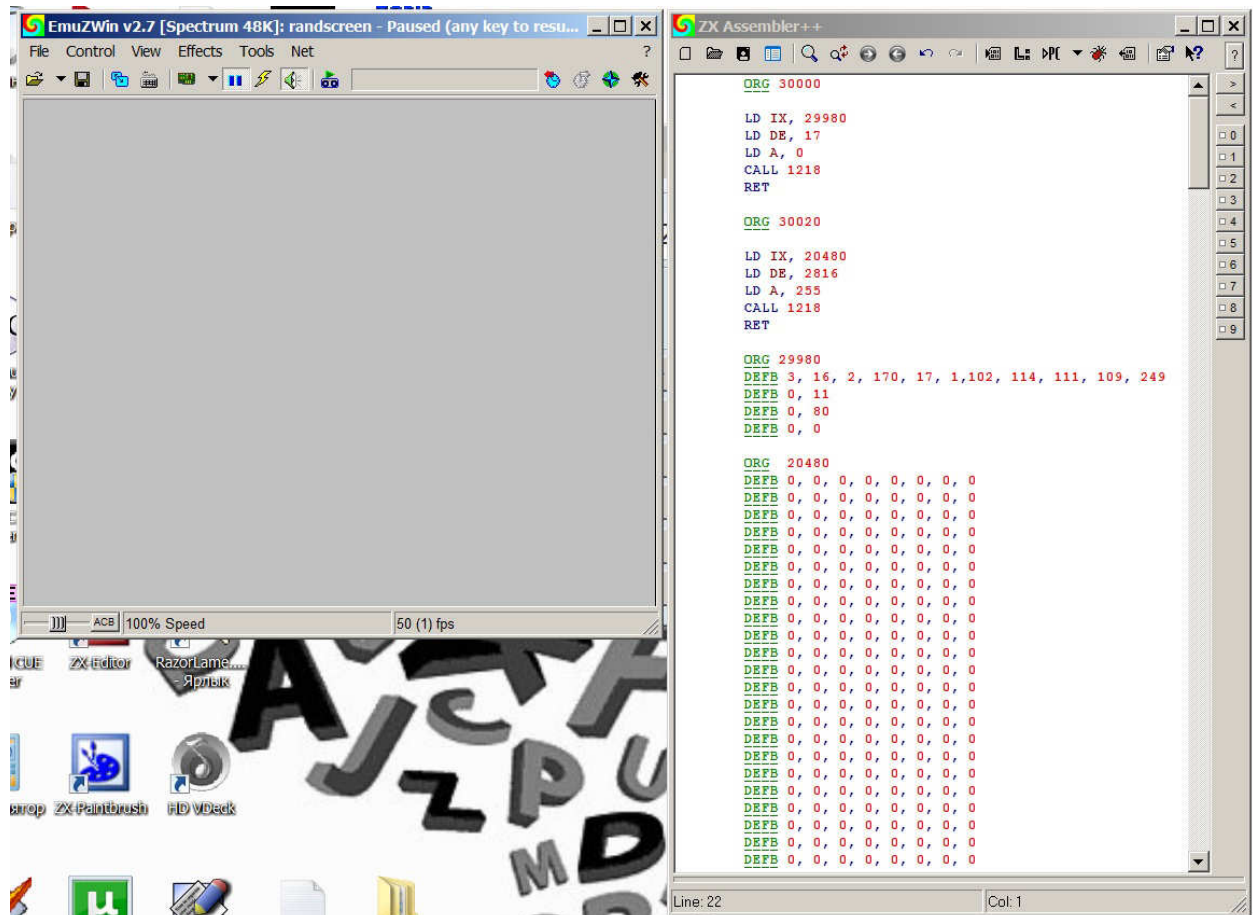


Рис. 2902. Полуфабрикат программы для окончательного создания *.tzx файла.

Открыв эту программу в Realspectrum, вы увидите тестовую картинку в нижних строках.

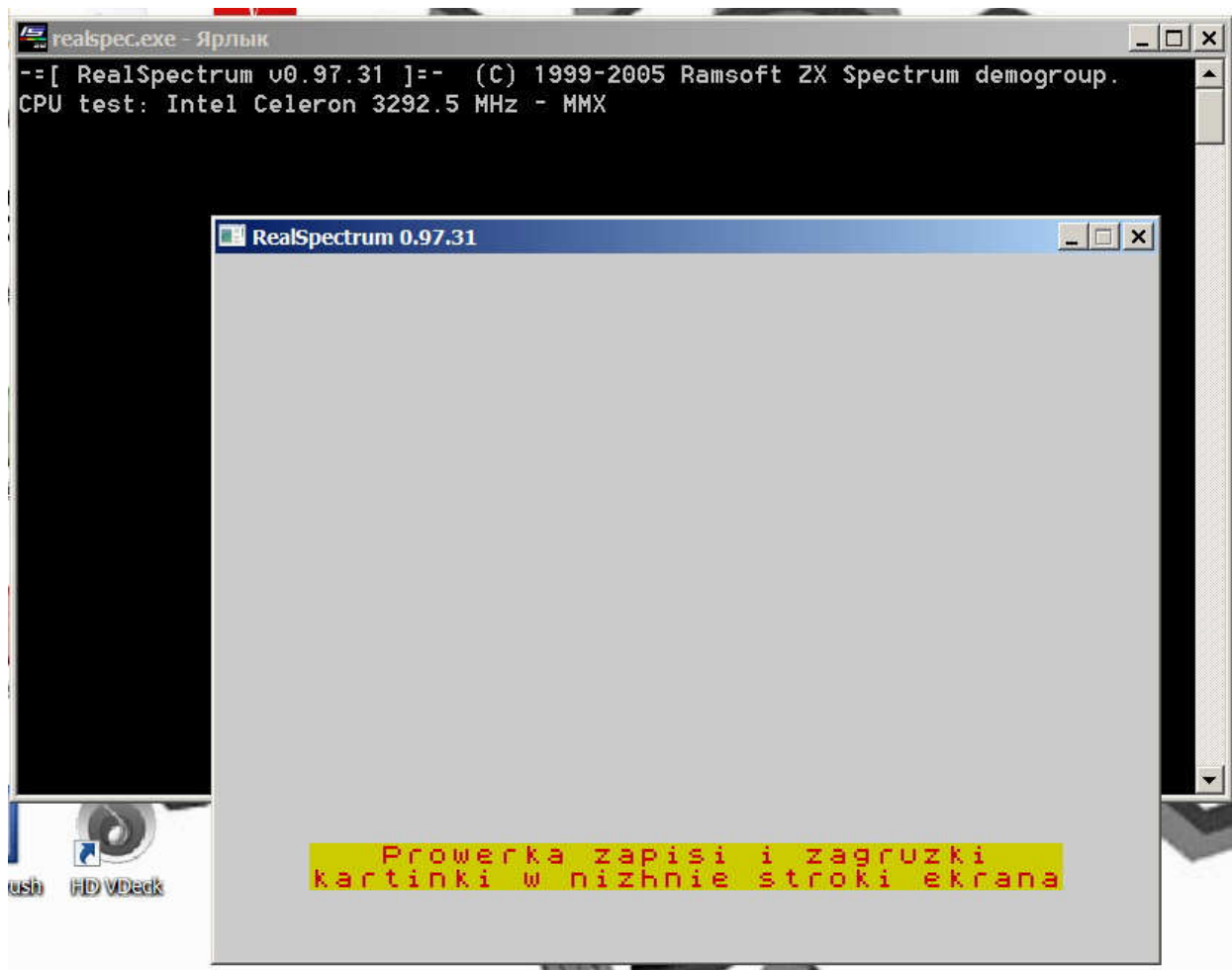


Рис. 2903. Проявление картинки при открытии слепка памяти*.z80 в эмуляторе Realspectrum.

Создайте удобным способом "randscreen.tzx".

Теперь проверим готовую программу на совместимость заголовка с данными, и наличия графики в нижних строках картинки. Запустим для проверки в Spectaculator по LOAD ""CODE: PAUSE 0. Выскочит цветной заголовок с длинным именем файла, а внизу начнет прорисовываться тестовая картинка.

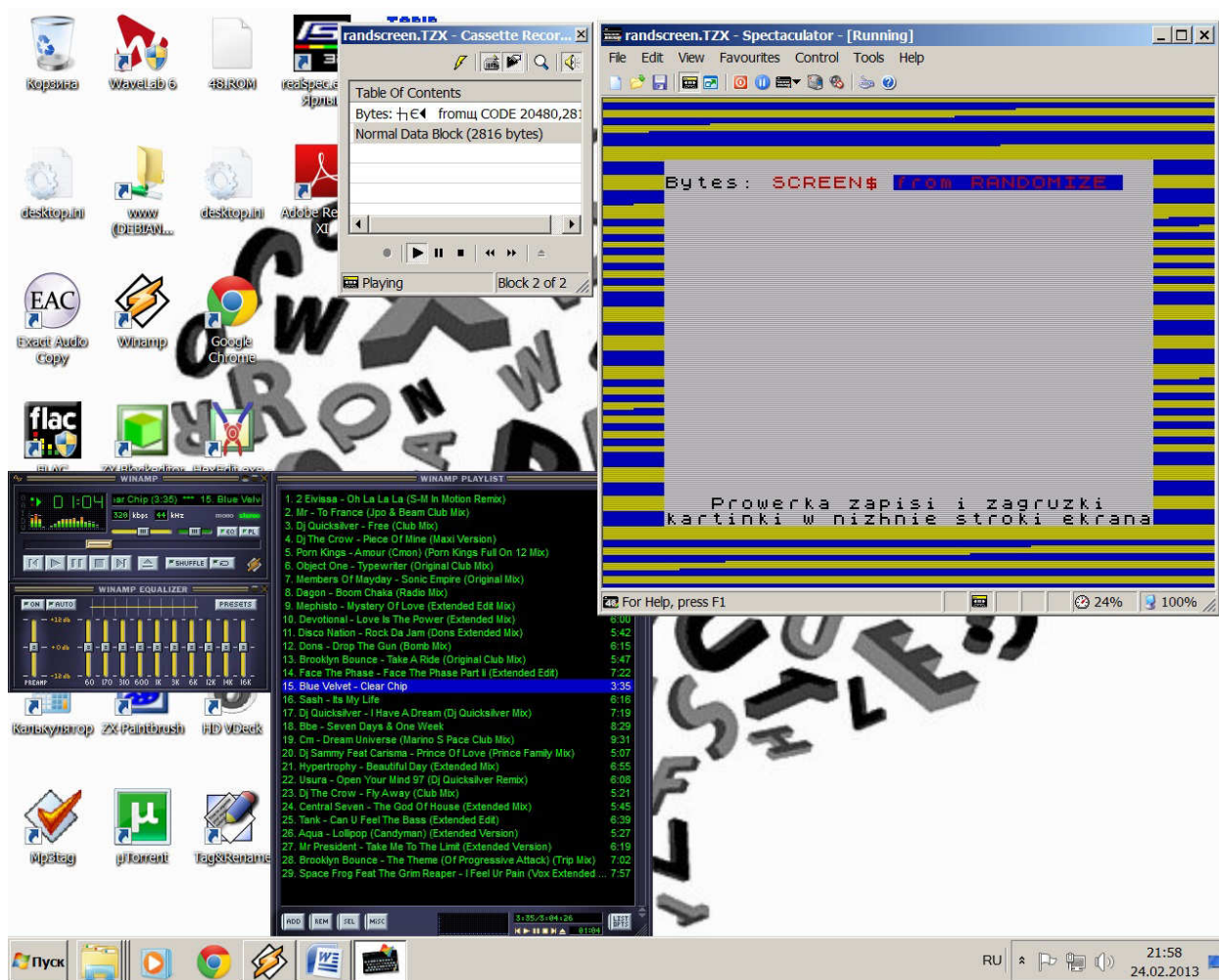


Рис. 2904. Процесс загрузки картинки с нижними строками в область экрана в Spectaculator'e при проверке.

После прорисовки программа будет ожидать нажатия любой клавиши, а затем нижние строчки сотрутся, и выскочит знакомое ОК, 1.

ЧАСТЬ 3.

Работы по модификации ПЗУ эмулятора ZX-Spectrum.

Глава 1.

Настройка эмулятора для изменения данных ПЗУ.

Краткое содержание: формат *.rom, подключение к эмулятору альтернативного ПЗУ

В настоящем спектре, область ПЗУ для большинства пользователей была запретной зоной, чем-то недостижимым и неизменным. В эпоху эмуляторов ПЗУ это всего-лишь файл с расширением *.rom. Практически во всех эмуляторах его можно изменять сторонними программами, а затем подсовывать в качестве прошивки. В некоторых эмуляторах даже есть опция по изменению данных в ПЗУ в реальном времени. Достаточно снять только галочку в настройках. Давайте попробуем поэкспериментировать с прошивкой Спектрума, но для начала настроим эмулятор. Испытывать модернизированную прошивку будем в Spectaculator'e. В этом эмуляторе основное ПЗУ находится в самой программе. Изменять исполняемый файл мы не будем, чтобы по неосторожности не повредить установленный эмулятор. В Spectaculator'e, есть функция присоединения альтернативной прошивки. При установке опция отключена, но в любой момент ее можно включить.

Первым делом, в папку, с установленным эмулятором (по умолчанию это *C:/Program Files/spectaculator.com/Spectaculator7*) поместите файл прошивки «48.rom». Предварительно перед экспериментами сделайте копию ПЗУ, с которой будете проводить опыты. Нужно это для того, чтобы после повреждений, при неудачных модернизациях, стирать файл и записывать новый.

Начнем с дополнительной настройки Spectaculator'a. Откройте эмулятор нажмите *CTRL+Y* (или кнопку «Tools», а в открывшемся меню «Options»). Откроется окно «Spectaculator options». В этом окне нажмите кнопку «Advanced», и внутри окошка будут следующие настройки:

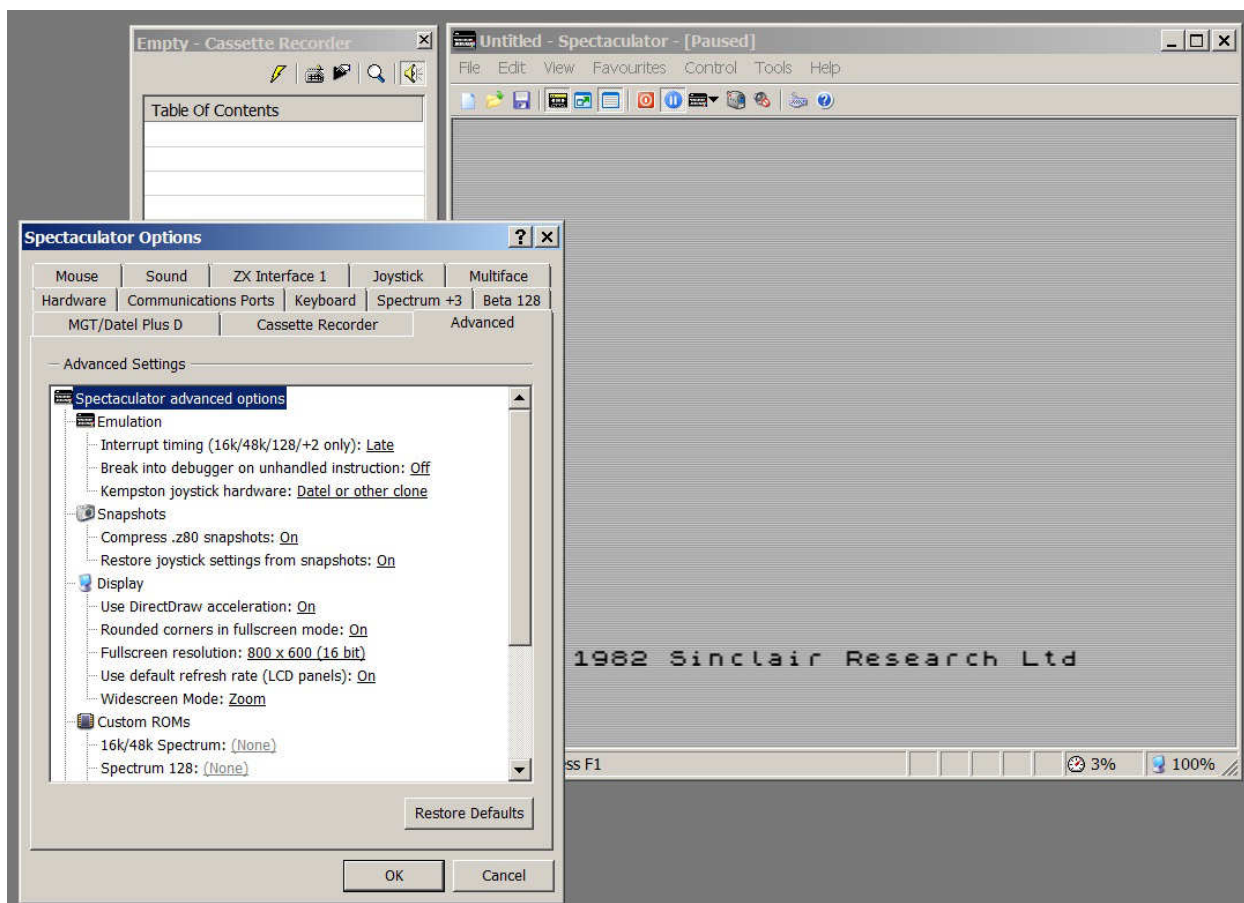


Рис. 3100. Окно настроек «Advanced» эмулятора Spectaculator.

Найдите опцию «Custom ROMs» и нажмите на «16k/48k Spectrum». Вместо надписи (None) откроется окошко с мигающим курсором для ввода имени альтернативной прошивки. Справа за окном будет кнопка с тремя точками:

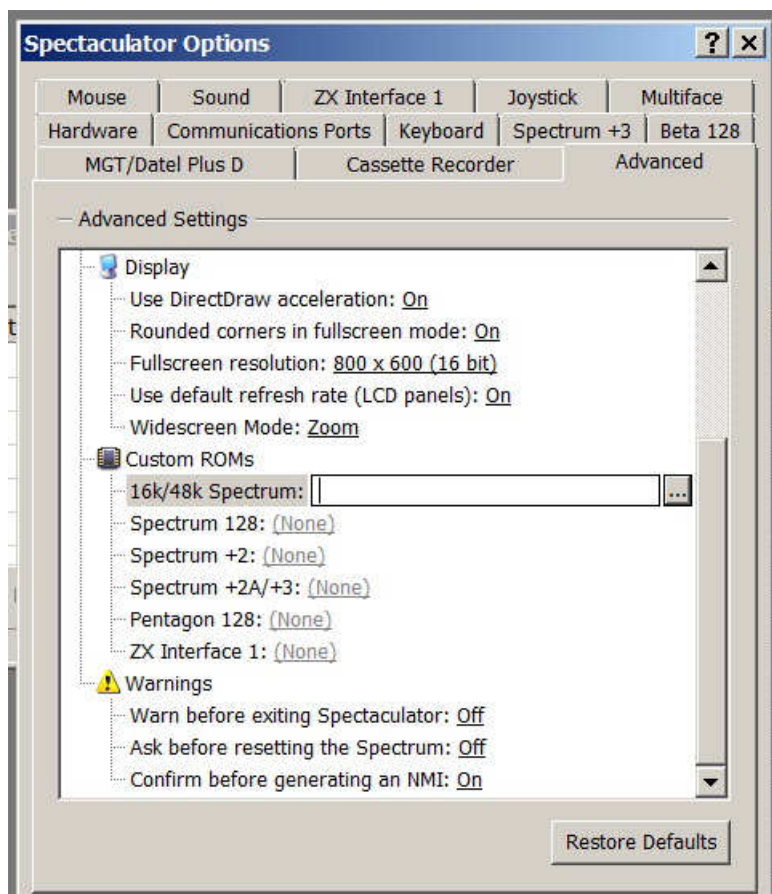


Рис. 3101. Настройка пути к альтернативному файлу ПЗУ.

Нажмите на нее и откроется еще одно окно «Open». В нем укажите путь к файлу альтернативного ПЗУ:

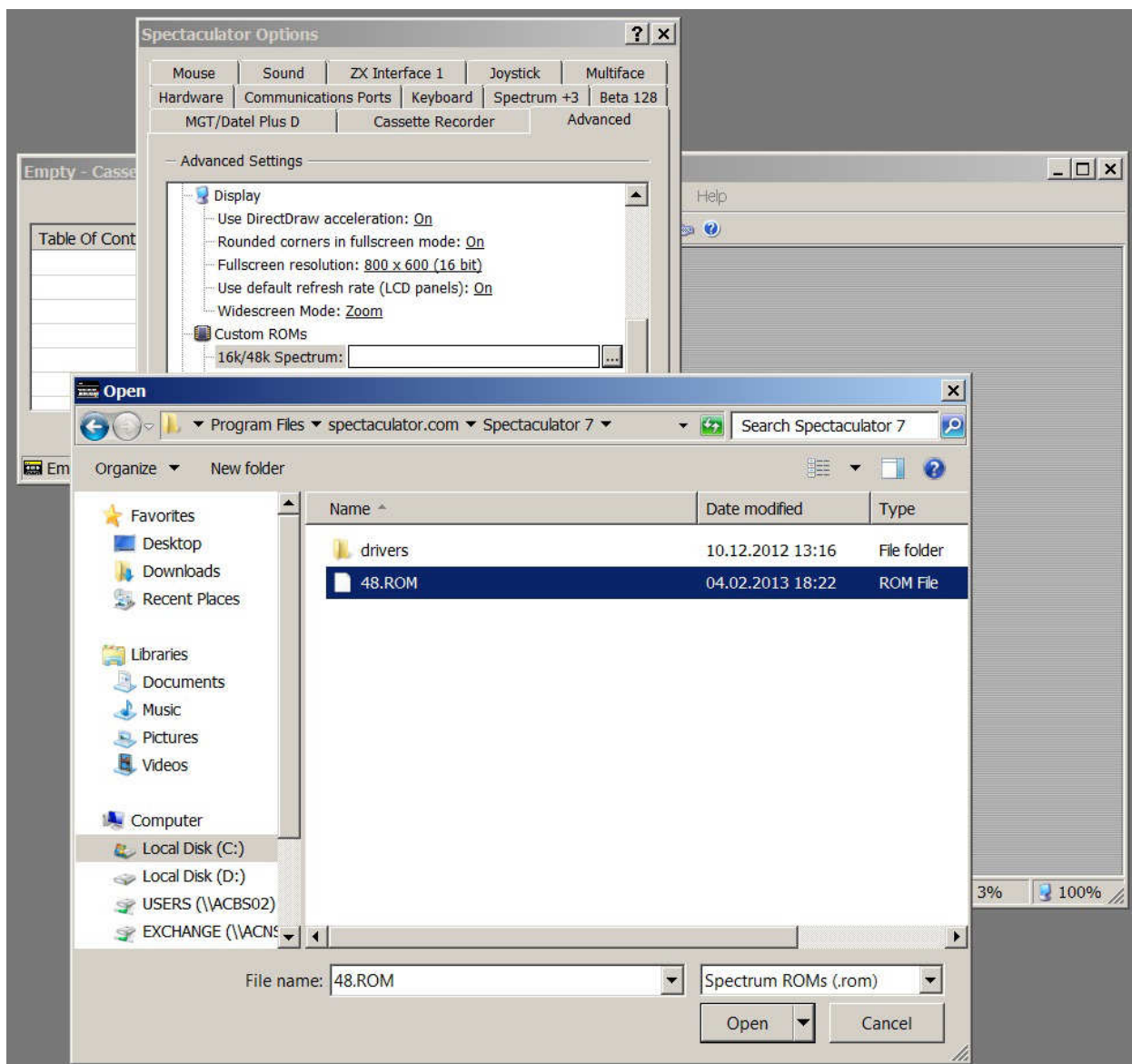


Рис. 3102. Выбор .rom файла альтернативного ПЗУ для эмулятора Spectaculator.

Нажмите «Open», окно закроется, и в пустой строке пропишется выбранный путь к альтернативному ПЗУ:

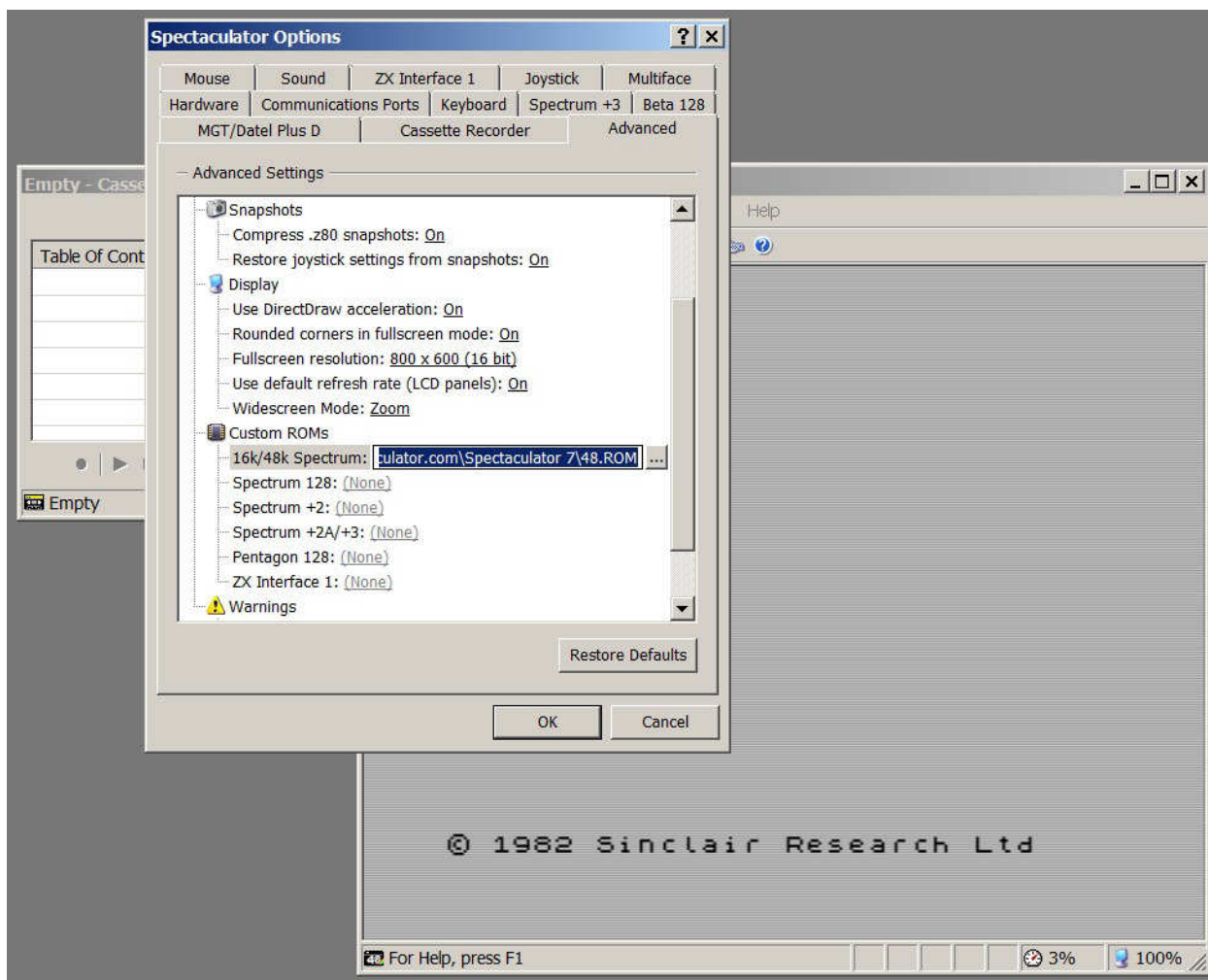


Рис. 3103. Путь к альтернативному файлу ПЗУ.

Нажмите кнопку «OK». Окно настроек закроется, а эмулятор автоматически перезагрузится, подключившись к выбранному вами ПЗУ.

Глава 2. Модернизация встроенного шрифта и текстовой информации в ПЗУ.

Краткое содержание: редактор Hex Edit основы работы, изменение текстовых данных ПЗУ, принципы русификации Спектрума.

Настроив эмулятор нужным образом, приступим к редактированию данных в ПЗУ. Для работы можно взять любой шестнадцатичный редактор файлов. Я предлагаю самый простой и бесплатный Hedit.

Давайте начнем с самого простого, и попробуем изменить текстовую информацию нескольких байт ПЗУ. Например, изменить текст приветствия в нижней части экрана:

(C) 1982 Sinclair Research Ltd.

Затем запустим эмулятор с модернизированной прошивкой, посмотрим на результат.

Помните, что во-избежании повреждения ПЗУ меняйте данные только байт на байт, не залезая ниже отведенного пространства, и только те данные, назначение которых знаете. Если длина текста, например 5 байт, значит, новый текст должен быть не более 5 байт в этой отведенной области. Если собираетесь модифицировать подпрограммы ПЗУ, то следует помнить, что программы для спектрума, на эмуляторе с самодельно модифицированной прошивкой, вряд ли запустятся.

Откройте Hedit. (я использовал v.4.0). Для удобства я переключаю на просмотр адресов в десятичном режиме. Это можно сделать, нажав в программе «Alt+D» или кнопкой с картинкой «FF/255». Откройте редактируемый файл. Для этого в редакторе нажмите «CTRL+O». Откроется стандартное окно «Open», в котором выберем путь к файлу «48.rom» в папке Spectaculator. Нажав кнопку «Open», в редакторе появится файл «48.rom» вот в таком виде:

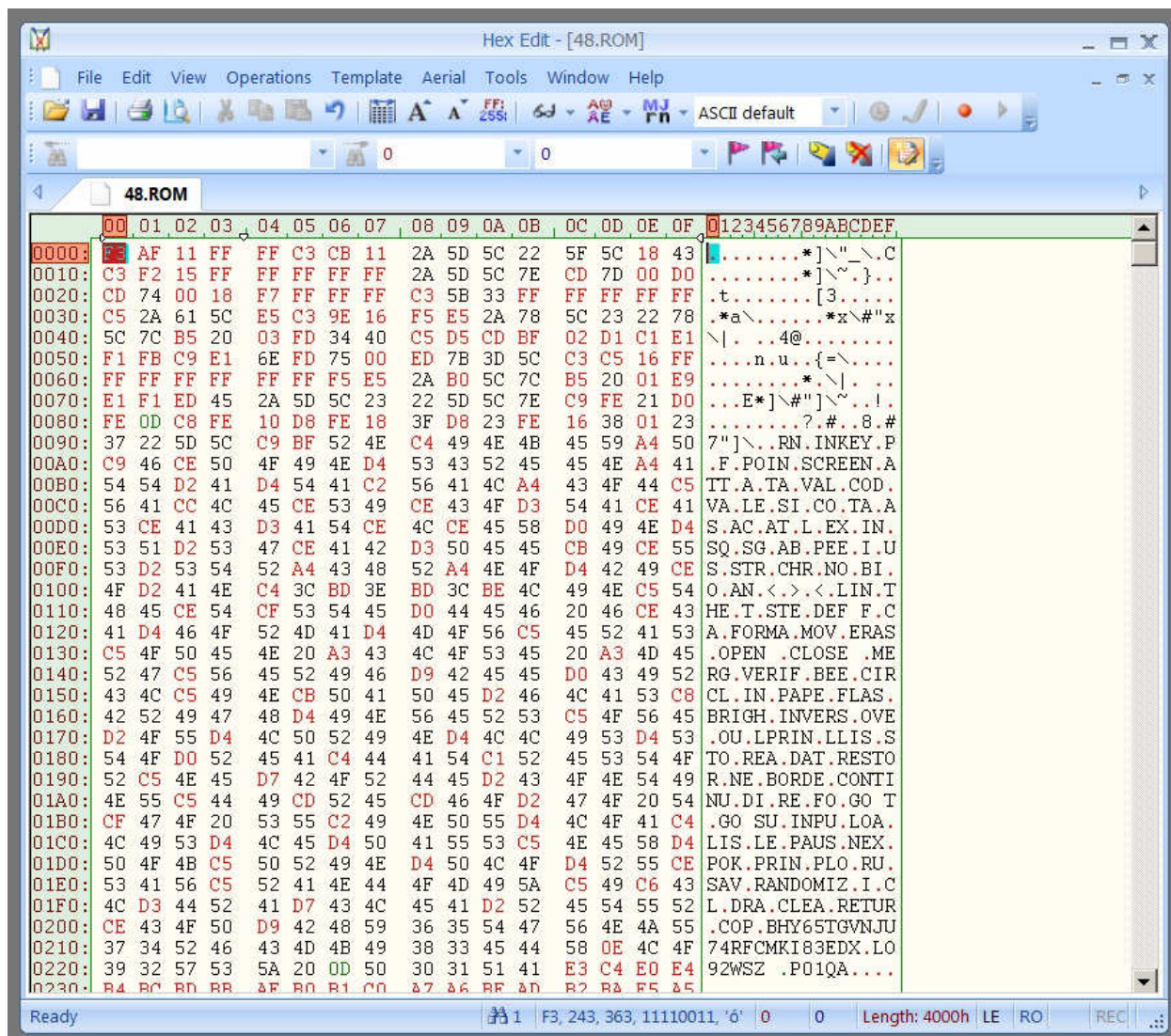


Рис. 3200. Общий вид ПЗУ ZX-Spectrum в шестнадцатиричном редакторе Hex Edit.

В данном случае работа упрощается, так как смещение в редакторе будет совпадать с настоящими адресами ПЗУ Спектрума. Сразу бросаются в глаза команды BASIC с искаженными последними буквами.

Давайте перейдем сразу к делу и найдем строку «© 1982...» Достаточно найти 1982, как последовательность текста. В редакторе нажмите «CTRL+F». Откроется окно поиска «Find». В нем переключитесь на вкладку «Text», в опции «Directions» переключите на кнопку «Down», и в окне наберите последовательность символов «1982».

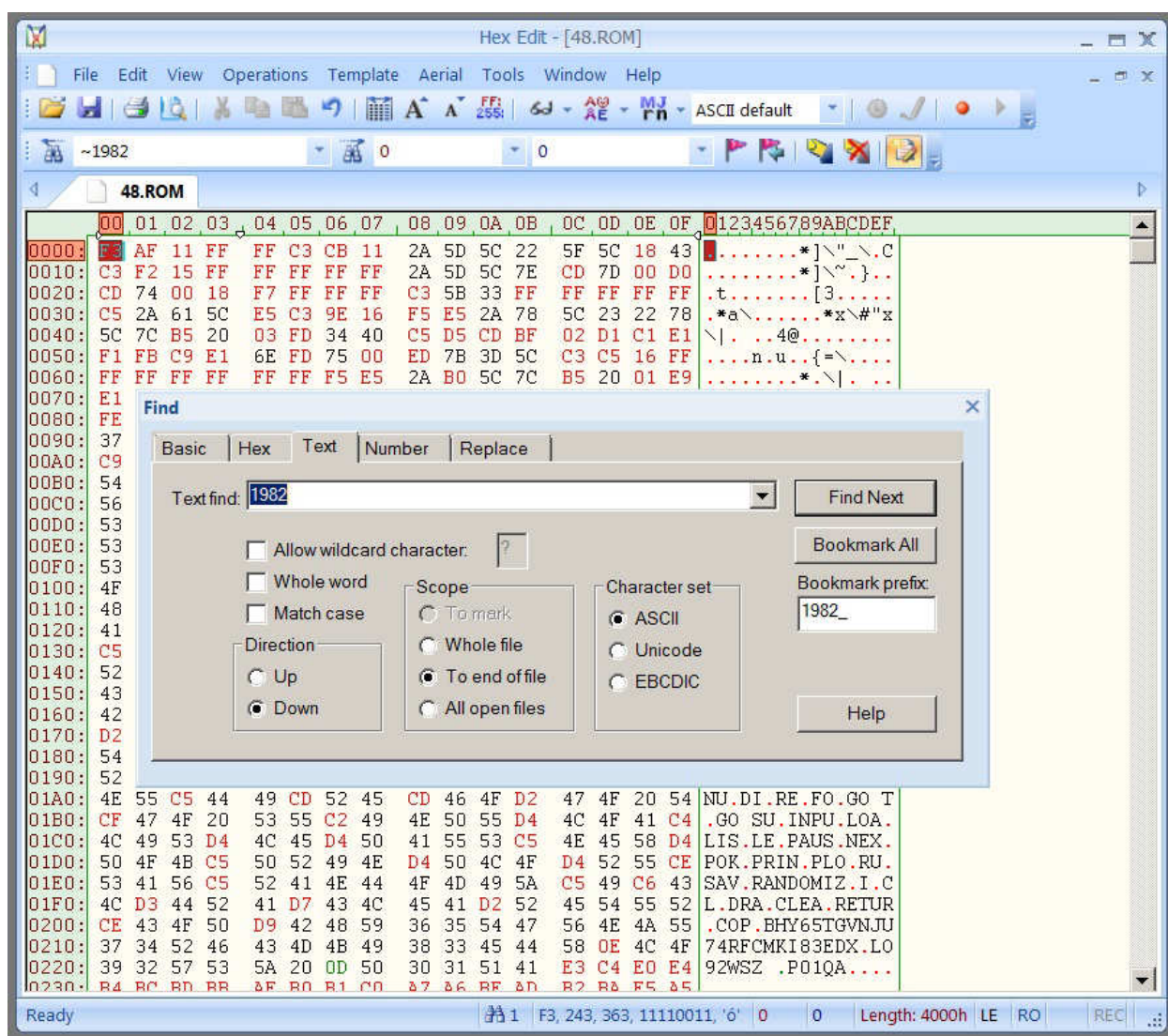


Рис. 3201. Редактор Hex Edit. Поиск нужной текстовой информации в файле ПЗУ.

Теперь нажмите кнопку «Find Text», и редактор выделит первую последовательность символов по адресу 1541, а дальше виднеется «Sinclair Research Lt.» Это как раз то что надо. Закрываем окно поиска и проматываем нужную строку чуть выше. Как видим, выше этой строки находятся тексты всех сообщений об ошибках BASIC системы:

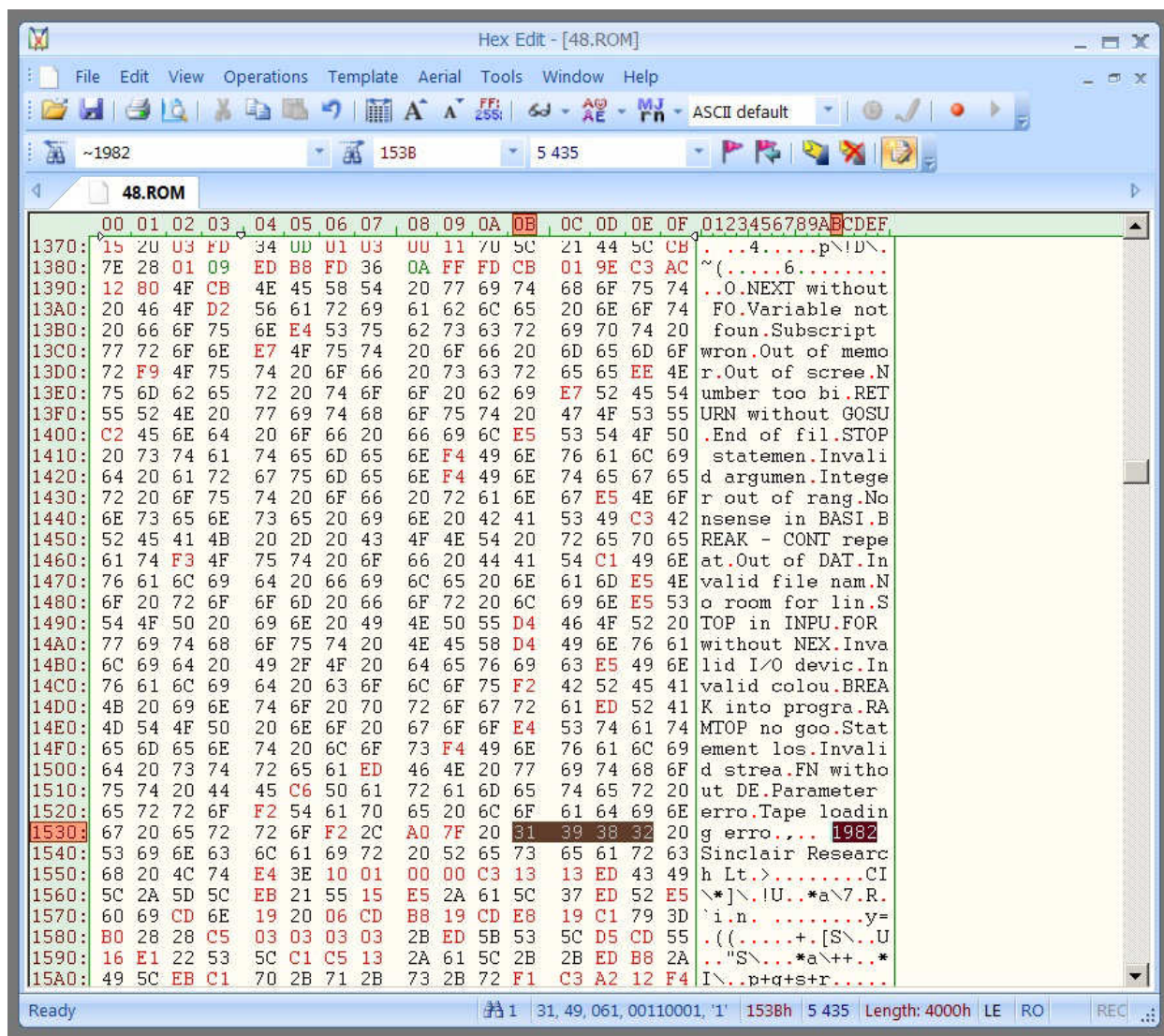


Рис. 3202. Редактор Hex Edit. Найденная цепочка символов «1982».

Теперь, начиная с единички, вместо текста приветствия, аккуратно введите «Prowerka Izmenennogo PZU » (с пробелом на конце). Должно получиться так:

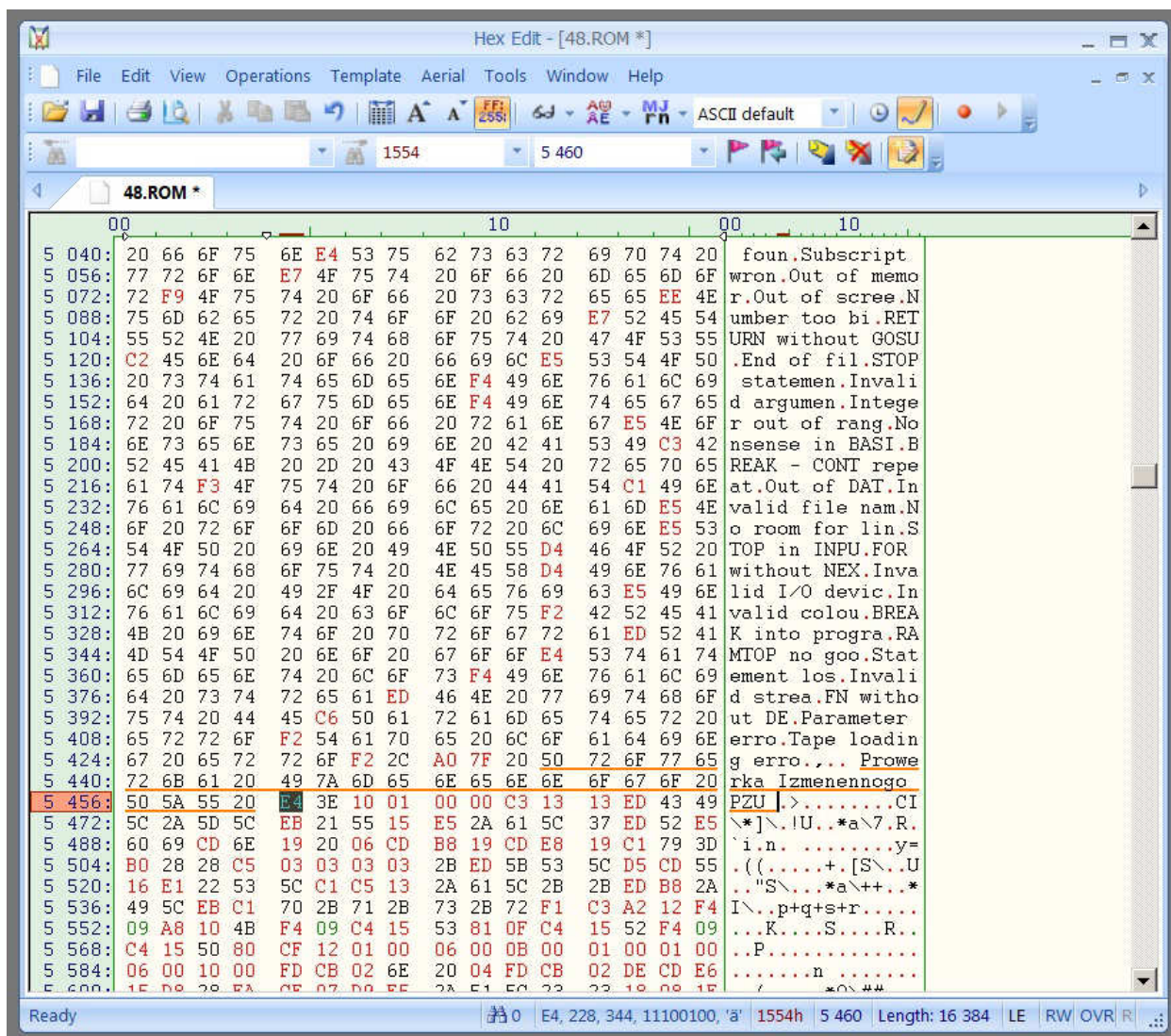


Рис. 3203. Редактор Hex Edit. Редактирование текстовой информации в ПЗУ ZX-Spectrum 48.

Теперь нажмите «**CTRL+S**» или кнопку с дискетой, и изменения в файле сохранятся.

Выйдем из редактора и испытаем измененное ПЗУ на эмуляторе. Открыв эмулятор, вместо привычного текста на заставке, мы увидим следующее:

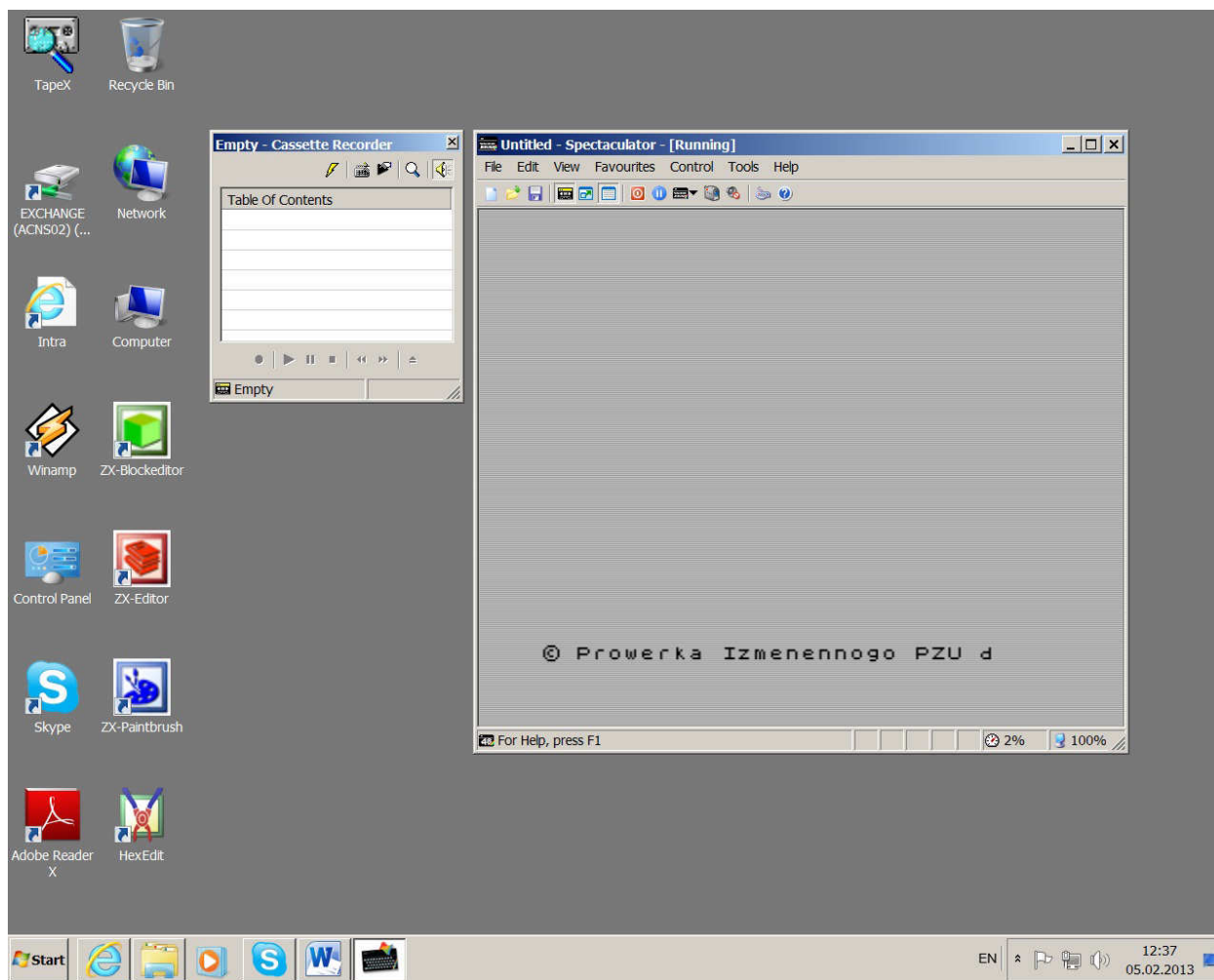


Рис. 3204. Тестирование эмулятора EmulZWin с измененным ПЗУ.

Можно зайти в BASIC, и проверить его работоспособность. Все должно быть в порядке. Нажав Reset, мы снова увидим измененное приветствие. Вы можете попробовать подключить это ПЗУ к другим эмуляторам, и попробовать запустить в них. Результат должен быть идентичным.

Теперь попробуем русифицировать букву «i» на «и» и записать ее в ПЗУ. Чтобы вычислить адрес размещения набора символов, обратимся к переменной **CHARS**, которая располагается по адресам 23606 и 23607. В ней находятся значения 0 и 60, что будет означать 15360. Но если мы посмотрим область, то никакого намека на буквы не найдем. Там все забито значением «255», а начинается шрифт с адреса 15616, символом пробел. Действительный адрес набора размещен на 256 байт нижеуказанного в переменной. Если приглядется, то можно понять, что первые 32 символа контрольные управляющие коды, и поэтому рисунков для них не существует, хотя место в памяти предусмотрено. Вычислить адрес размещения текущего рисунка символа достаточно просто по формуле:

$$\text{Адрес Символа} = 15616 + (n - 32) * 8$$

Находим в таблице символов код буквы «i». Это будет «105». Следовательно, рисунок буквы «i» будет размещен по адресу $15616 + (105 - 32) * 8 = 16200$.

Теперь создадим рисунок маленькой буквы «и» в шестнадцатиричном формате. Он будет выглядеть так: 0, 0, 44, 4C, 54, 64, 44, 0.

Снова откроем в Hex Edit файл «48.rom», найдем адрес 16200 и введем 8 байт нового рисунка буквы «i»:

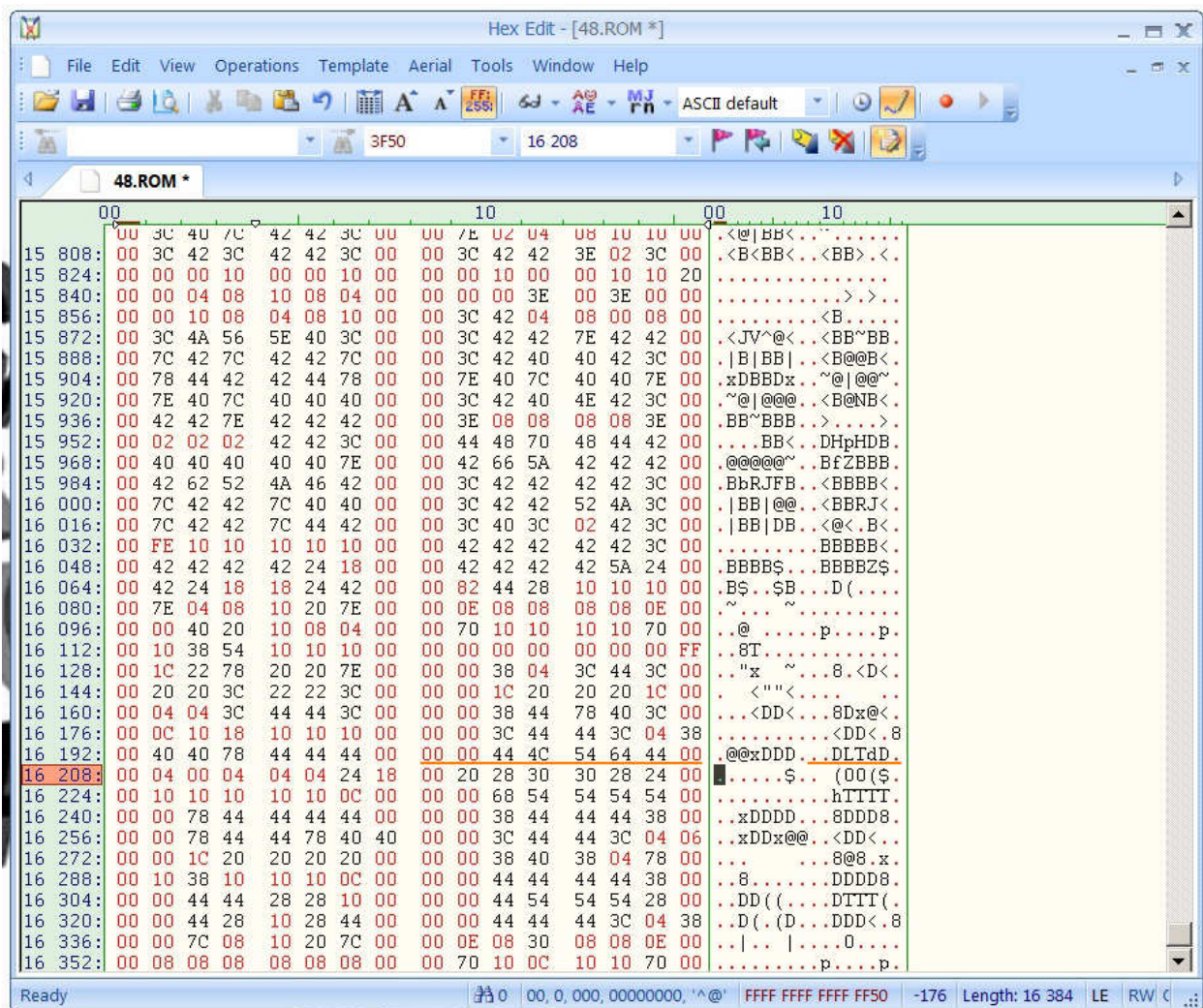


Рис. 3205. Редактор Hex Edit. Байты модифицированного символа буквы «i».

Сохраняем файл, закрываем редактор, и снова запускаем эмулятор с измененным ПЗУ, понаблюдать изменения. А они заметны сразу, во время включившейся заставки. Латинская буква «i» заменилась русской «и». Теперь нажмите клавишу «I» несколько раз и после выдачи команды **ИМПУТ**, вы увидите несколько русских букв «и», а значит, эксперимент удался:

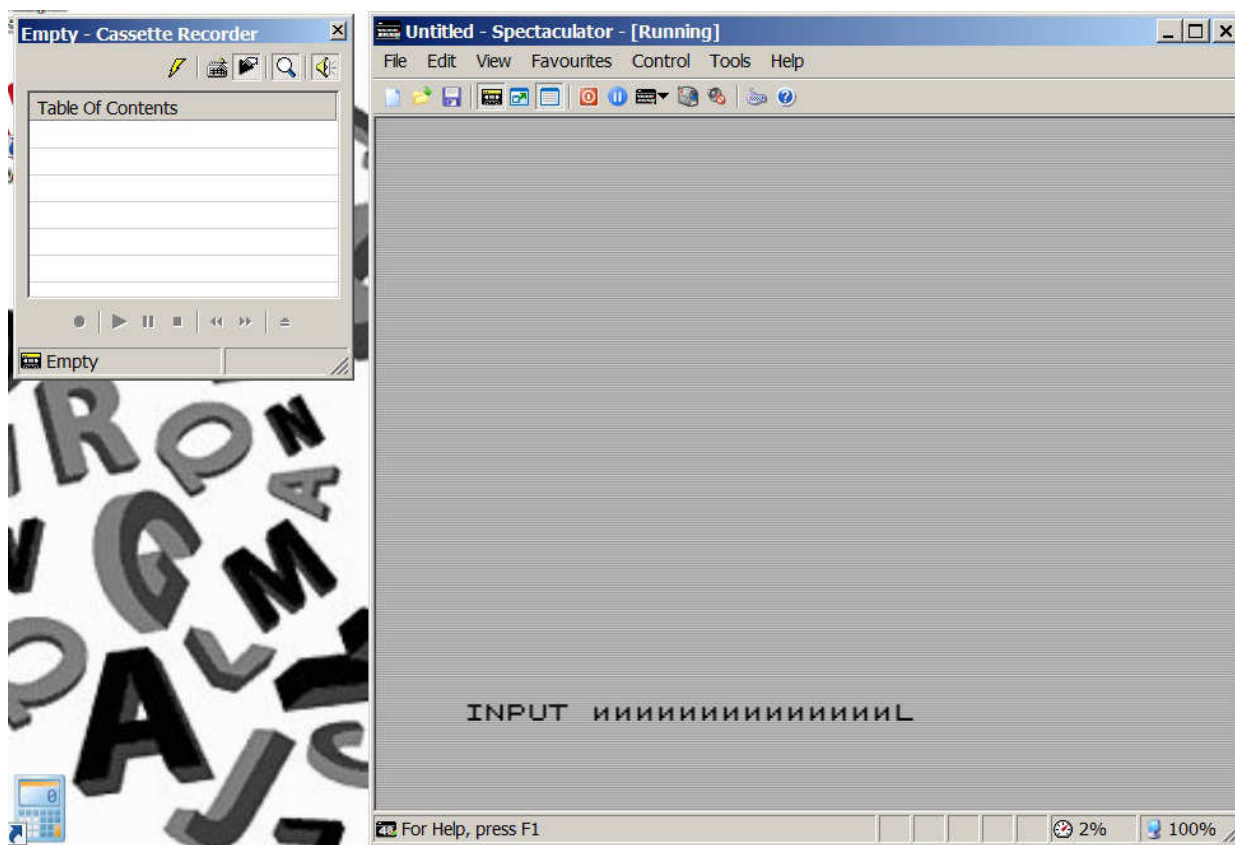


Рис. 3206. Проверка печати модифицированного символа "и" в эмуляторе Spectaculator.

Глава 3. Изменение команд и сообщений интерпретатора BASIC.

Краткое содержание: таблица команд, маркировка символов, изменение команд и сообщений в ПЗУ, принципы русификации команд BASIC.

Когда мы редактировали текст заставки, то сразу заметили, что все команды и сообщения, которые попались в ПЗУ, с поврежденной буквой на конце. Если внимательно изучить эти искаженные коды, то видно что они имеют последовательность, схожую с нормальными кодами символов. Чтобы наглядно понять, как изуродованы буквы, переведем несколько кодов символов на калькуляторе в двоичный вид, а затем переведем символы, которые должны там стоять. При беглом сравнении видно, что коды букв искажены добавлением 7-го бита к их коду. Условно говоря к каждому символу просто прибавлено число «128». Поэтому при изменении команд и сообщений об ошибках следует это учитывать. Нужно это для того, чтобы программа знала окончание слова в команде, или тексте сообщения, потому что слово не имеет привязки к определенной ячейке памяти.

Учитывая такой подарок, сделанный разработчиками компьютера, можно поэкспериментировать с переводом команд BASIC на русский язык. Например, в области ПЗУ стоят рядом три команды: `RUN`, `SAVE` и `RANDOMIZE`. Наиболее кратко можно перевести `RUN` это ПУСК, а `SAVE` - ЗАПИСЬ, но букв в первой команде всего 3, а во 2-й 4, а для русского перевода надо 4 и 6. В данном случае все просто, попробуем позаимствовать пару букв у длинной команды `RANDOMIZE`.

Вернемся к практике. Сотрите ПЗУ, измененное в прошлой главе, и замените на оригинальное. Для этого просто сотрите испорченный файл и замените новым с тем же именем. Если сомневаетесь, можно проверить подключилось ли новое ПЗУ, запустив Spectaculator.

Откройте оригинальное ПЗУ в Hex Edit. Найдем область таблицы команд. Она начинается с адреса 00149. Найдем команду «*RUN*» и вместо нее введем «*PUSK*». Для этого к коду клавиши «К» (75) прибавим «128». Получим 203, что будет СВ в шестнадцатиричном виде. Добавим «*PUS*» в поле символов, и «СВ» в поле чисел, сразу за «*S*», который превратится в точку. Следом изменим команду «*SAVE*» превратив в «*ZAPIS*», а букву «*X*», эквивалентную мягкому знаку представим в виде 88+128=216 (HEX=D8). Сохраните файл нажатием «*CTRL+S*»:

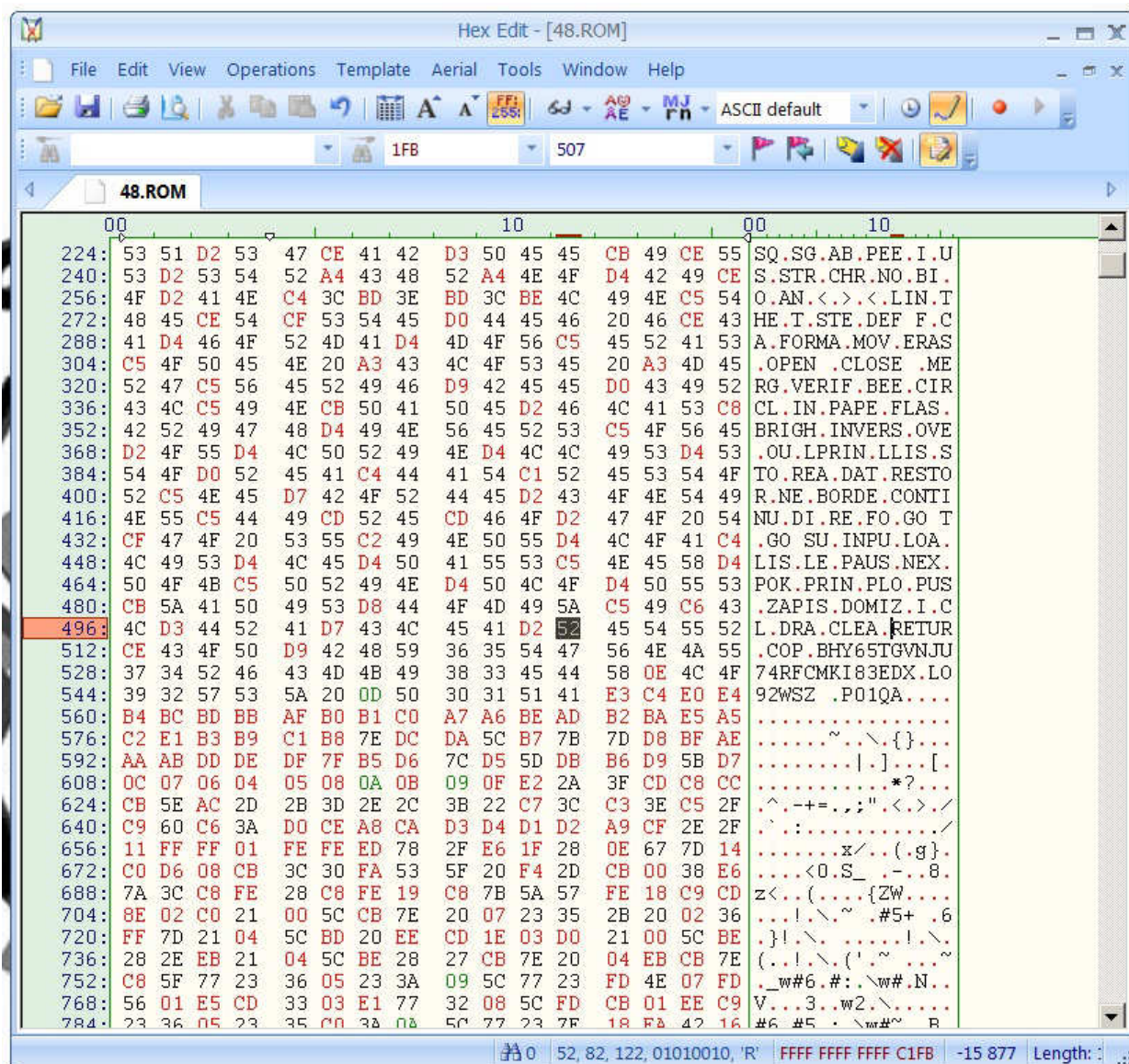


Рис. 3300. Редактор Hex Edit. Русифицированные команды *PUSK* (*RUN*) и *ZAPISX* (*SAVE*)

Теперь прокрутим данные до адреса 5000 и попробуем русифицировать сообщения интерпретатора BASIC. Они записаны точно таким же образом, как команды, с плавающим адресом и прибавлением к коду символа «128». Изменим, к примеру, сообщение «7 RETURN without GO SUB» на «7 WOZWRAT без WYZOWA» и «9 STOP statement» на «9 Operator STOP». Лишние символы замените модифицированным пробелом с кодом «A0»:

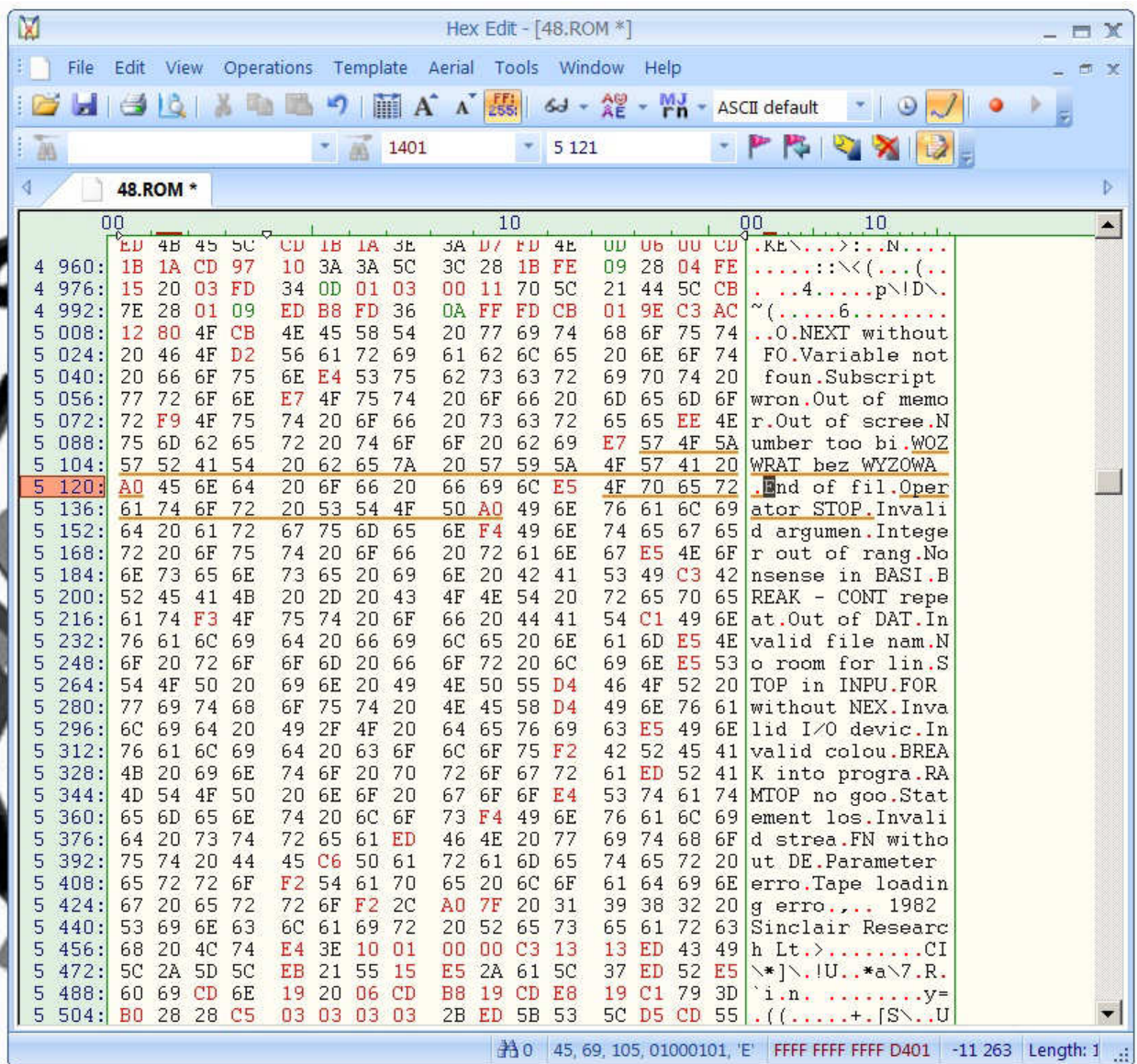


Рис. 3301. Редактор Hex Edit. Процесс русификации сообщений интерпретатора BASIC.

Сохраним измененный файл и выйдем из редактора. Зайдем в Spectaculator и попробуем ввести такую программу на BASIC:

```
1 LIST
2 RETURN
3 SAVE «RusBASIC»
4 RUN
```

Первые две строки вводятся нормально, а в строке 3 нажав букву S вместо «SAVE» напечатается «ZAPISX», а вместо «RUN» увидим «PUSK». В итоге мы увидим на экране:

```
1 LIST
2 RETURN
3 ZAPISX "RusBASIC"
4 PUSK
```

Теперь запустим программу и увидим наше русифицированное сообщение об ошибке:

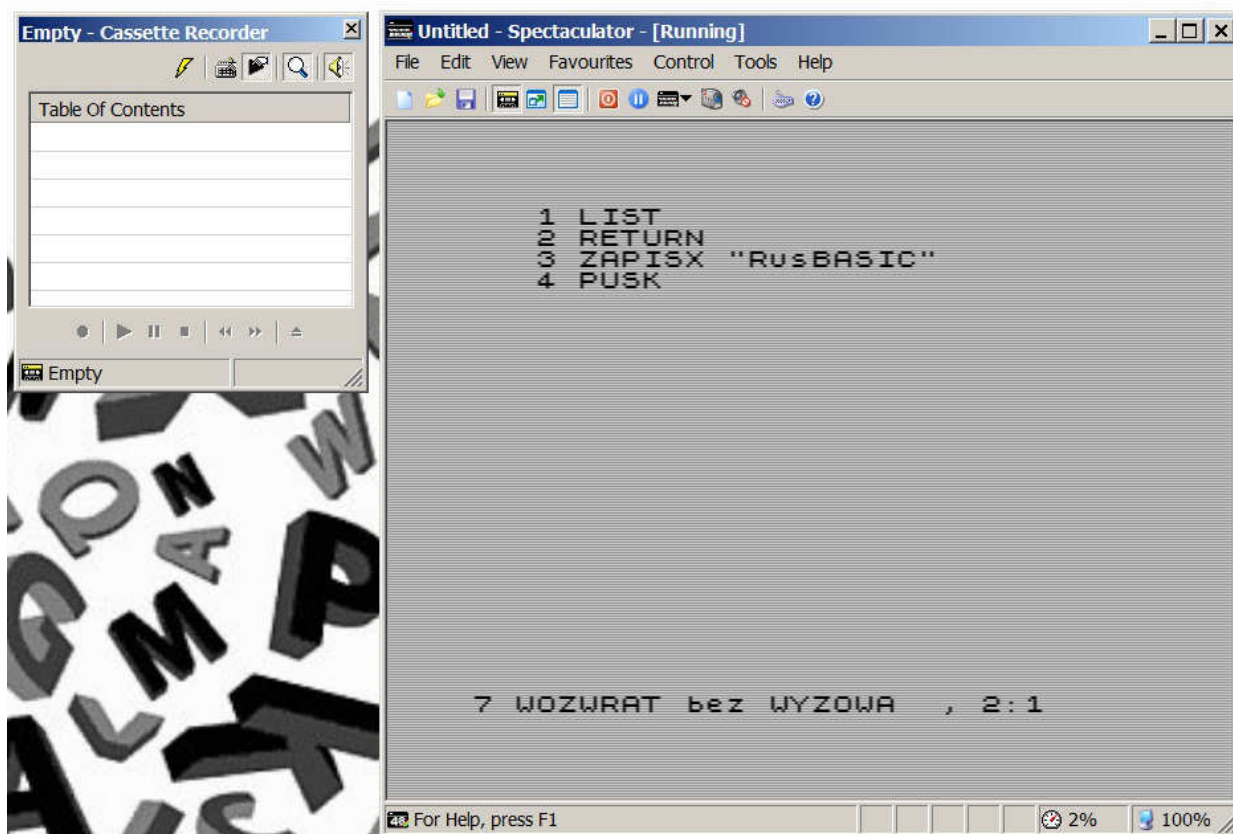


Рис. 3302. Проверка ПЗУ с измененными командами и сообщениями на эмуляторе Spectaculator.

Глава 4.

Простая модификация программ в ПЗУ и изменение цвета полос загрузки.

Краткое содержание: детальное изучение подпрограмм загрузки данных ПЗУ, исследование последовательности цветов полосок на рамке, модификация команды BEEP.

В первой части книги рассматривали простейшую модификацию загрузчика, путем частичного замещения процедуры из ПЗУ в области ОЗУ. Теперь рассмотрим более детально, все точки, где можно сменить цветовые полосы, подобрав наиболее экзотическое сочетание цветов. В общем, важные будут те места, где идет изменение 3-х младших битов байта, перед выводом в порт «254». Давайте определим эти важные точки:

.....

1381 AND A, 32

1383 OR A, 2

1385 LD C, A

.....

1439 LD A, C

1440 XOR A, 3

1442 LD C, A

.....

1532 LD C, A

1533 AND A, 7

1535 OR A, 8

1537 OUT (254), A

.....

Наиболее важным фрагментом будет самый последний. AND A, 7 отсекает лишние биты, оставляя биты цветов, красно/голубой для пилот-тона и желто/синий для данных. Командой OR 8 добавляется бит управления выводом на магнитофон.

При приеме пилот тона, в регистре A, к адресу 1533, чередуясь, приходят значения «253» и «2» (верха и низа сигнала). Числа проходят такие преобразования:

253 → AND 7 → 5 → OR 8 → 13 → голубая полоска
2 → AND 7 → 2 → OR 8 → 10 → красная полоска

При загрузке данных на этот же адрес приходят значения «222» и «33»:

222 → AND 7 → 6 → OR 8 → 14 → желтая полоска
33 → AND 7 → 1 → OR 8 → 9 → синяя полоска

Быстрое чередование смены цветов рамки дают эффект бегущих полосок на экране. Становится понятно, если изменить программу, перед выводом рамки, то можно изменить цвет полосок. Недостаток в том, что нужно уложиться в 4 байта, заменив полностью или изменив значения AND 7 и OR 8.

Наиболее интересных комбинаций полосок можно добиться, замещением команды OR 8 на ADD A, n, где n любое число (0-7). Желательно, чтобы полученное значение не выходило за пределы 16. Прибавляя к действующему значению регистра A число, тем самым сдвигаем цвета рамки. Программу следует изменить так:

AND A, 7
ADD A, n

(E6, 07, C6, n)

Давайте попробуем изменить программу в ПЗУ и посмотреть на результат. Например так:

1533 AND A, 7
1535 ADD A, 7

Для этого в Hex Edit нужно ввести в адрес 1535 и 1536 значения C6 и 07. Откройте файл «48.rom» в редакторе, измените, значения и сохраните изменения. Теперь откроем Spectaculator и попробуем загрузить какую-нибудь программу, из ранее написанных. При загрузке на рамке будут видны красивые переливы цветов:

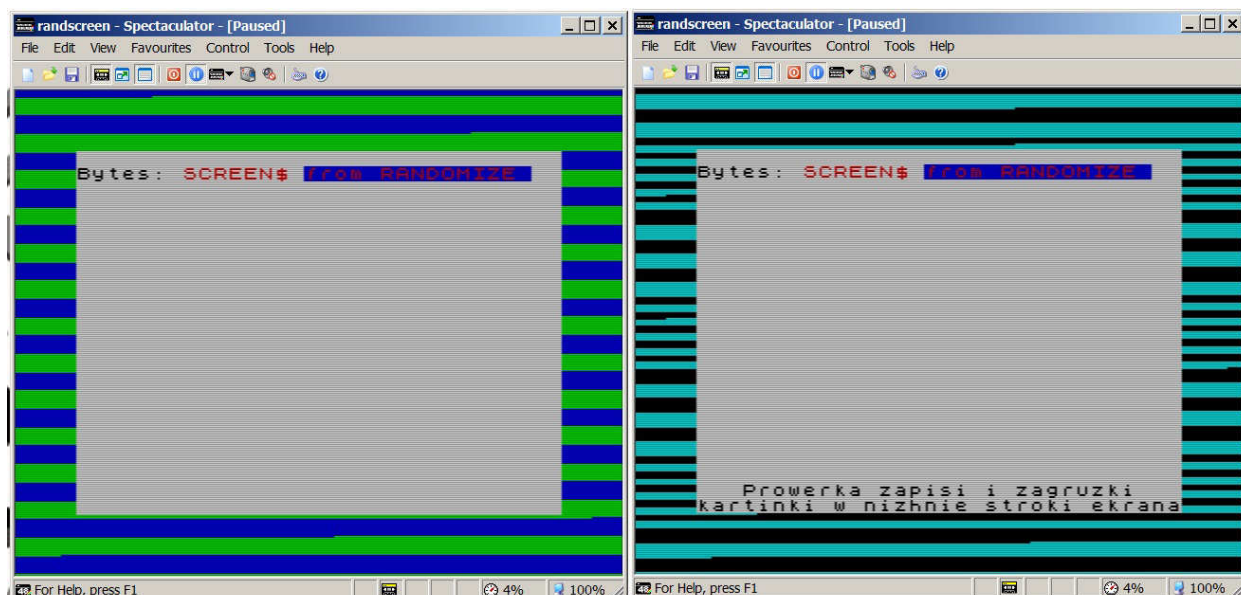


Рис. 3400. Проверка измененного ПЗУ с необычными цветами полосок на рамке.

Подставляя числа и наблюдая за изменениями цветов, можно прийти к выводу, что почти все цвета сигнала верхней и нижней половины имеют определенную последовательность и одинаковый шаг между кодами цветов верха и низа сигнала.

Установив значение AND в 0, загрузка будет происходить незаметно, без полос на экране, так как все полосы будут одного цвета, заданного в операторе ADD.

При AND 1, полосы будут всего 2-х цветов, как для пилот-тона, так и для данных. Оператором ADD будет задаваться меньшее значение цвета. Например, при ADD 1, полосы будут сине-красные для пилот-тона и данных.

При AND 02 чередование цветов будет через один, например черный-красный, синий-фиолетовый.

Давайте составим таблицу значений, полученных в ходе лабораторных опытов с изменениями значений AND и ADD:

Простые комбинации цветов, где пилот-тон и данные одного цвета:

	AND 0	AND 1	AND 2	AND 4
ADD 0	Черный	Черный/синий	Черный/красный	Черный/зеленый
ADD 1	Синий	Синий/красный	Синий/фиолетовый	Синий/голубой
ADD 2	Красный	Красный/фиолетовый	Красный/зеленый	Красный/желтый
ADD 3	Фиолетовый	Фиолетовый/зеленый	Фиолетовый/голубой	Фиолетовый/белый
ADD 4	Зеленый	Зеленый/голубой	Зеленый/желтый	Зеленый/черный
ADD 5	Голубой	Голубой/желтый	Голубой/белый	Голубой/синий
ADD 6	Желтый	Желтый/белый	Желтый /черный	Желтый /красный
ADD 7	Белый	Белый/черный	Белый/синий	Белый/фиолетовый

Сложные комбинации, где пилот-тон и данные разных цветов:

	AND 5	AND 6	AND 7
ADD 0	Черный/голубой - синий/зеленый	Красный/зеленый - черный/желтый	Голубой/красный - синий/желтый
ADD 1	Красный/голубой - синий/желтый	Фиолетовый/голубой - белый/синий	Желтый/фиолетовый - красный/белый
ADD 2	Фиолетовый/желтый - красный/белый	Зеленый/желтый - черный/красный	Белый/желтый - фиолетовый/черный
ADD 3	Зеленый/белый - фиолетовый/черный	Голубой/белый - синий/фиолетовый	Черный/голубой - зеленый/синий
ADD 4	Голубой/черный - зеленый/синий	Желтый/черный - красный/зеленый	Синий/желтый - голубой/красный
ADD 5	Желтый/синий - голубой/красный	Белый/синий - фиолетовый/голубой	Красный/белый - желтый/фиолетовый
ADD 6	Белый/красный - желтый/фиолетовый	Черный/красный - зеленый/желтый	Фиолетовый/черный - белый/зеленый
ADD 7	Черный/фиолетовый - белый/зеленый	Синий/фиолетовый - голубой/белый	Зеленый/синий - черный/голубой

Команда XOR 3 по адресу 1440 будет регулировкой шага и сдвига комбинаций цветов сигнала данных. Некоторые комбинации могут дать интересное сочетание цветов:

	XOR 1	XOR 2
AND 7 ADD 0	Голубой/красный - фиолетовый/зеленый	Голубой/красный - белый/черный
AND 7 ADD 1	Желтый/фиолетовый - зеленый/голубой	Желтый/фиолетовый - черный/синий
AND 7 ADD 2	Белый/зеленый - голубой/желтый	Белый/зеленый - синий/красный
AND 7 ADD 3	Черный/голубой - желтый/белый	Черный/голубой - красный/фиолетовый
AND 7 ADD 4	Синий/желтый - белый/черный	Синий/желтый - фиолетовый/зеленый
AND 7 ADD 5	Красный/белый - черный/синий	Красный/белый - зеленый/голубой
AND 7 ADD 6	Фиолетовый/черный - синий/красный	Фиолетовый/черный - голубой/желтый
AND 7 ADD 7	Зеленый/синий - красный/фиолетовый	Зеленый/синий - желтый/белый

Но и тут мы можем выбирать только определенную последовательность цветов. В этом и заключается недостаток. Нельзя задать конкретные цвета для каждого вида сигнала, или например, сделать переливающуюся рамку. Чтобы это реализовать, придется модернизировать программу загрузки, и учесть некоторые факторы, чтобы не повредить ее. Иначе программа может потерять совместимость, и стандартные программы не будут загружаться в ОЗУ. Вклинивать настройки цветов рамки можно попробовать, например, вместо опроса клавиши **BREAK**.

Еще один пример модификации цветовых полос, приведем с командой **БЕЕР**, так как она тоже завязана на порт «254». Попробуем, например, сделать так, чтобы при звучании ноты по рамке бегали полосы как при загрузке. В ПЗУ программа команды **БЕЕР** (**BEER**) находится по адресу «00901».

Точно так же, как и с загрузкой данных, находим в программе команду **OUT** (254), А и видим такой фрагмент программы:

```

.....
0091 XOR 16
0093 OUT (254), A

```

Число 16, это 10000 в двоичном представлении. Для того чтобы установить цвет, одновременно с выводом звука, нужно просто добавить младшие биты цветов. Например, откройте Hex Edit и вместо XOR 16, по адресу 00992, поставьте XOR 17, и сохраните изменения.

Откройте Spectaculator и наберите **БЕЕР 1, 0**. Одновременно со звуком, снизу вверх быстро пробегут желтые полосы. Наберите **БЕЕР 1, 2**. Теперь полосы побегут уже вниз. Значит можно подобрать такие дробные ноты, чтобы во время звучания, полосы на рамке стояли.

Наберите **БЕЕР 1, 2, 4**, и вы увидите, как толстые желтые полосы практически застынут на месте. Тоже самое можно увидеть, набрав **БЕЕР 1, 5, 06**:

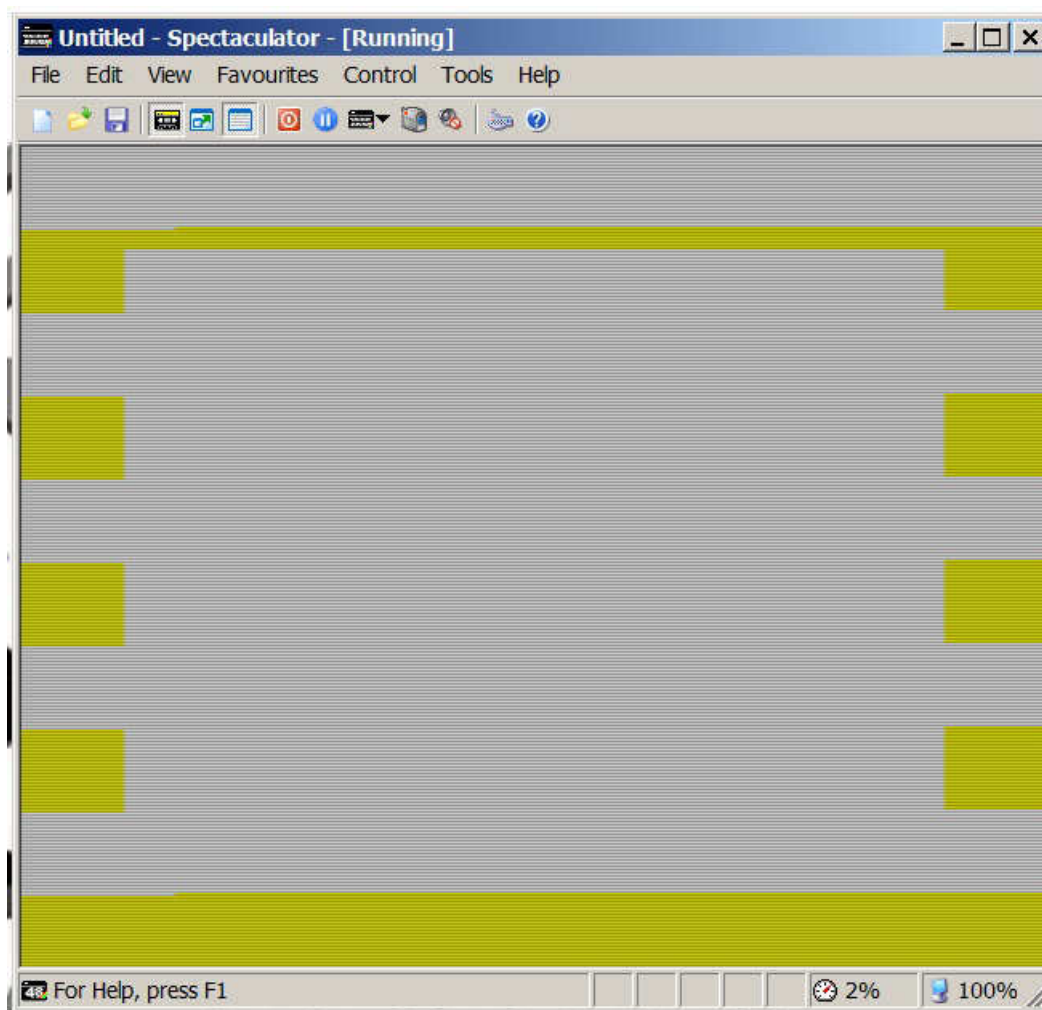


Рис. 3401. Модифицированная программа команды BEEP в процессе выполнения на эмуляторе.

Ниже приведены таблица значений XOR для разных цветов полосок:

	16	17	18	19	20	21	22	23
XOR	Белый	Желтый	Голубой	Зеленый	Фиолетовый	Красный	Синий	Черный

Заметим, что с возрастанием значений, цвета располагаются в обратной последовательности.

Глава 5.

Создание и запуск простейшей программы в ПЗУ.

Краткое содержание: создание файла с данными, редактирование и вставка данных в Hex Edit, принцип формирования собственного ПЗУ.

В прошлых главах мы рассмотрели, как русифицировать, менять графику, команды и даже редактировать некоторые подпрограммы в ПЗУ. Теоретически может возникнуть мысль вообще обнулить ПЗУ, и создать там свою программу, с адреса 0. Тогда при сбросе компьютер должен будет переходить к выполнению написанной программы. Только полноценной программы маленького размера все равно не получится. Все процедуры вывода на экран, и много других полезных функций придется писать самостоятельно. Давайте рассмотрим такую упрощенную программу:

START: LD B, 0


```

        LD A,247
        IN A, (254)
        RRA
        JR NC,BORDER1
        RRA
        JR NC,BORDER2
        RRA
        JR NC,BORDER3
        RRA
        JR NC,BORDER4
        RRA
        JR NC,BORDER5
        LD A,239
        IN A,(254)
        RRA
        JR NC,BORDER0
        RRA
        JR NC,CHISTYJ
        RRA
        JR NC,EXIT
        RRA
        JR NC,BORDER7
        RRA
        JR NC,BORDER6
        JR START
BORDER0: LD B, 249
BORDER7: INC B
BORDER6: INC B
BORDER5: INC B
BORDER4: INC B
BORDER3: INC B
BORDER2: INC B
BORDER1: INC B
        LD A, B
        OUT (254),A
        JR START
CHISTYJ: LD B, 0
        LD DE, 49151
        LD HL, 16384
CHISTYJ1: LD (HL), B
        INC HL
        DEC DE
        LD A, D
        CP 0
        JR NZ, CHISTYJ1
        LD A, E
        CP 0
        JR NZ, CHISTYJ1
        JR START
EXIT:

```

Данная программа по нажатию цифровых клавиш изменяет цвет рамки экрана. Чтобы не набирать данную программу вручную, можно ввести готовый набор кодов, который уже преобразован в шестнадцатичный вид:

```
06003EF7DBFE1F30291F30251F30211F301D1F30193EEFDBFE1F300E1F30191F302D1F30
071F300518D606F90404040404040478D3FE18C8060011FFBF21004070231B7AFE0020F87
BFE0020F318B1
```

Теперь создадим собственную прошивку. Откройте программу Hex Edit, и создайте новый чистый файл данных, длиной 16384 байта. Для этого в программе нажмите **CTRL+N**. Откроется окошко «*New File*» с настройками для создания будущего файла. В пункте «*Specified*» укажите размер создаваемого файла. В окне «*Decimal*» напишите 16384 байта. Под ним в окне «*Hex:*» автоматически выставится длина в шестнадцатичном формате. Это будет 4000:

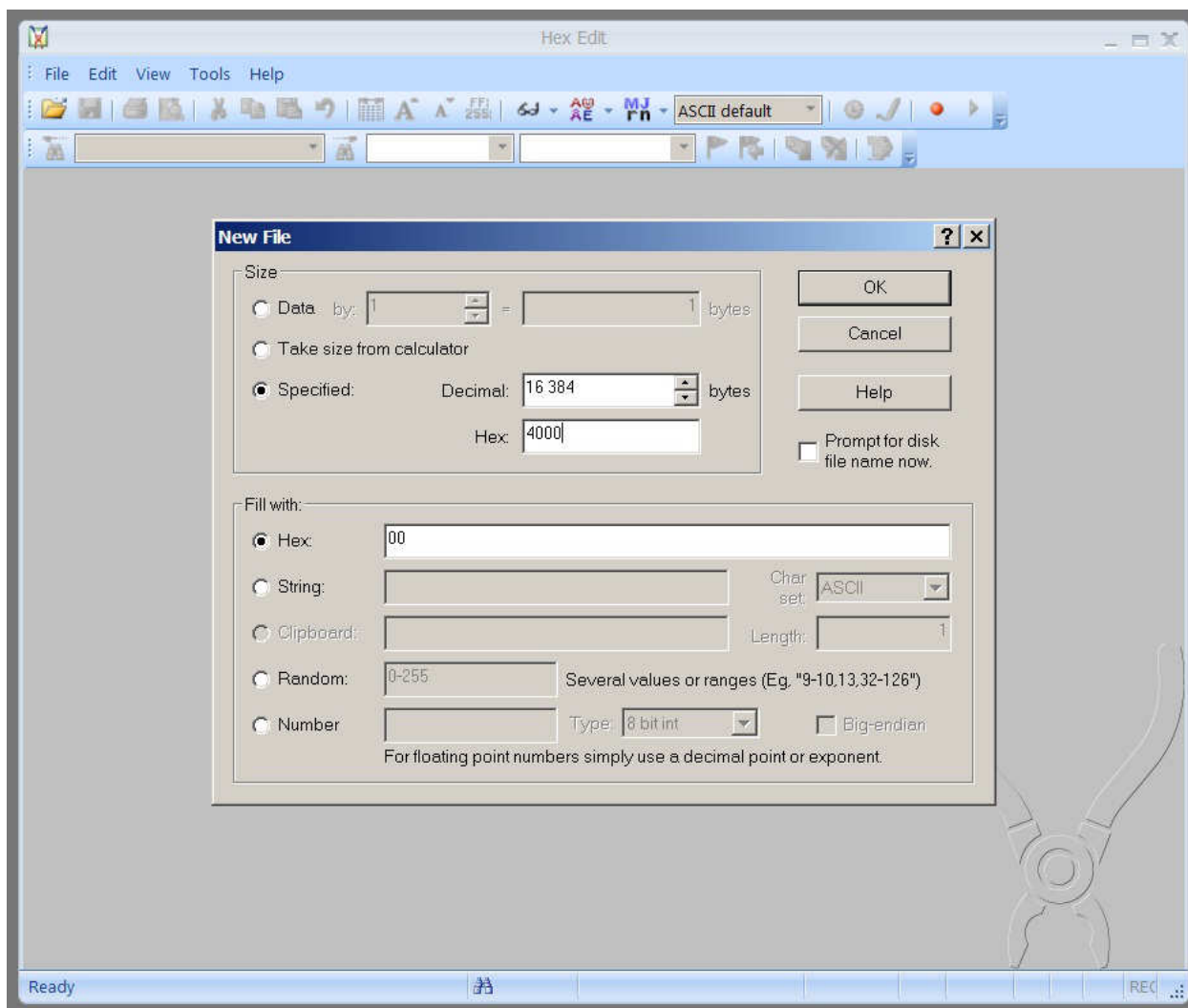


Рис. 3500. Редактор Hex Edit: Создание пустого файла длиной в 16384 байта.

Нажмите «OK» и файл, длиной 16384 байта, заполненный нулями, создан. Теперь по **CTRL+C** копируем блок кода с предыдущей страницы книги. Создаем еще один файл, только теперь переставим кнопочки из «*Specified*» в меню «*Data*». По умолчанию там будет стоять 1 байт. Теперь переходим в нижнюю секцию «*Fill with:*», а в окно «*Hex*» вставим данные, которые рассортируются точно по значениям. В «*Data*» длина скорректируется и отобразится 79 байт:

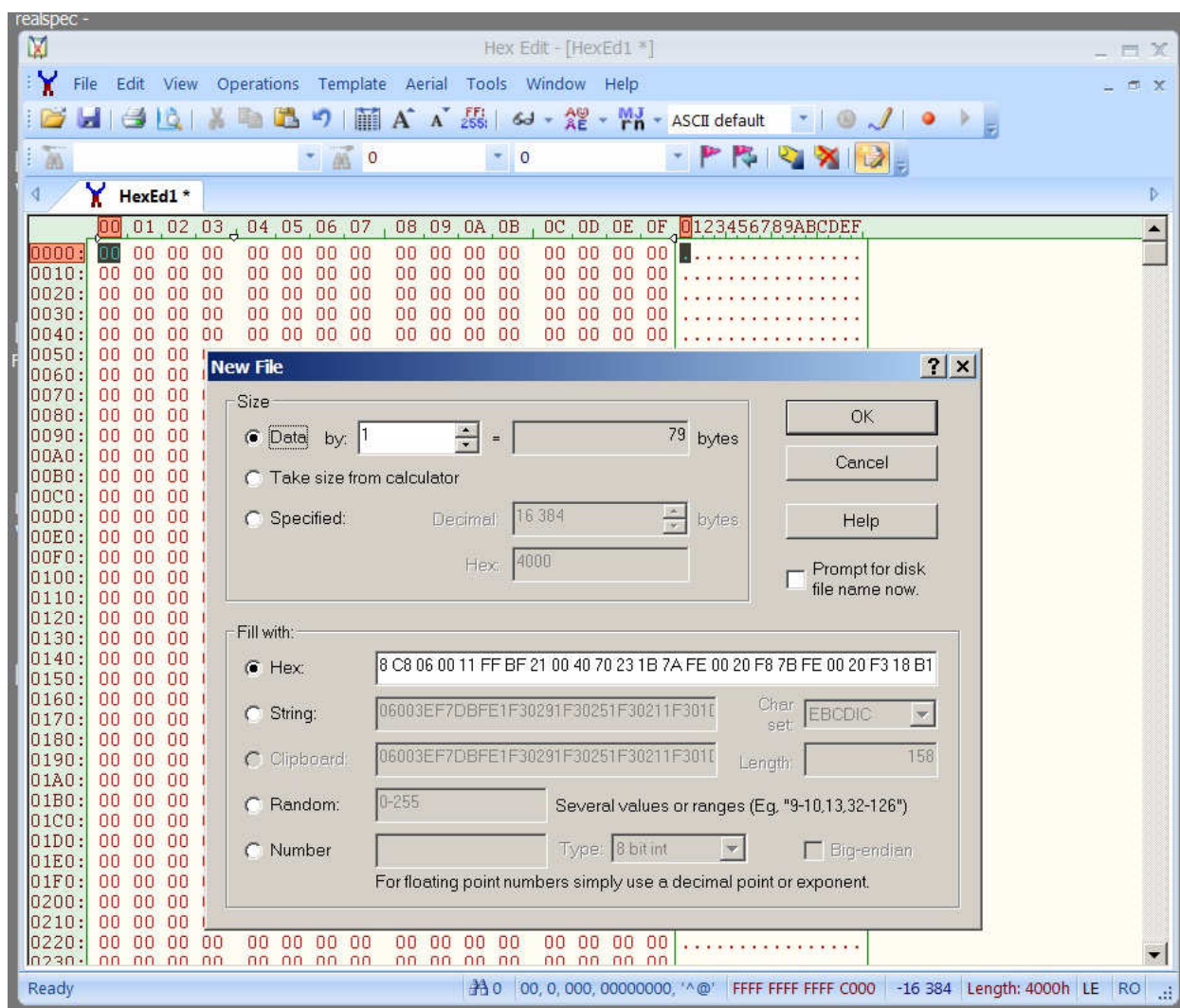


Рис. 3501. Редактор Hex Edit: Создание файла с преобразованием текста в шестнадцатичные данные.

Нажав «OK» создастся еще один файл с программой. Теперь копируем полученную таким образом программу в буфер по **CTRL+C**:

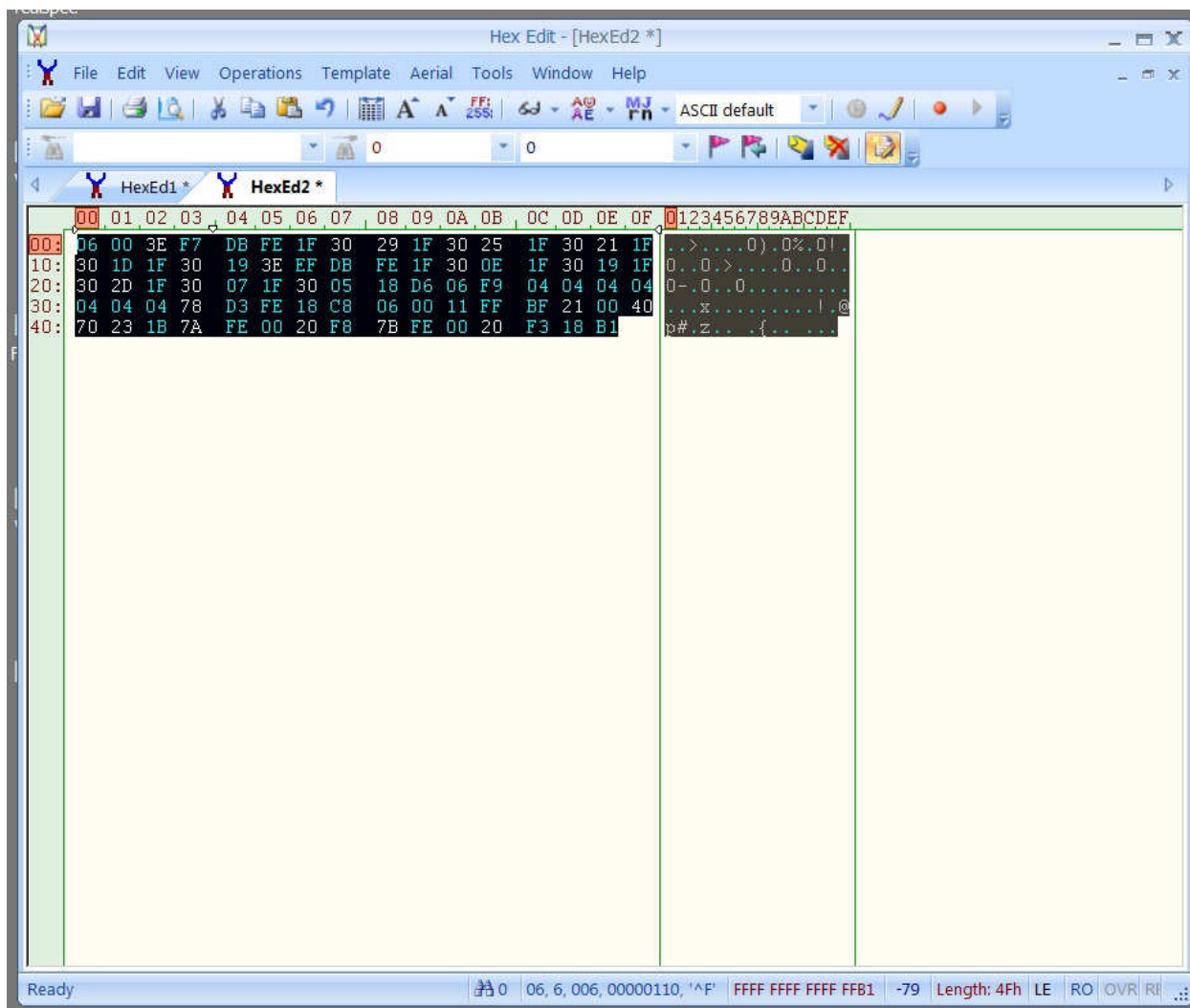


Рис. 3502. Редактор Hex Edit: Вырезание преобразованных шестнадцатичных данных для вставки.

Переходим в пустой файл, устанавливаем курсор в начало (если вы что-то делали до этого) и копируем туда данные. Перед копированием высочит запрос: «*Pasting in overwrite mode will overwrite data! Do you want to turn on insert mode?*». На запрос окна отвечаем «No», и новые данные вставятся в режиме замещения, не увеличивая длину файла:

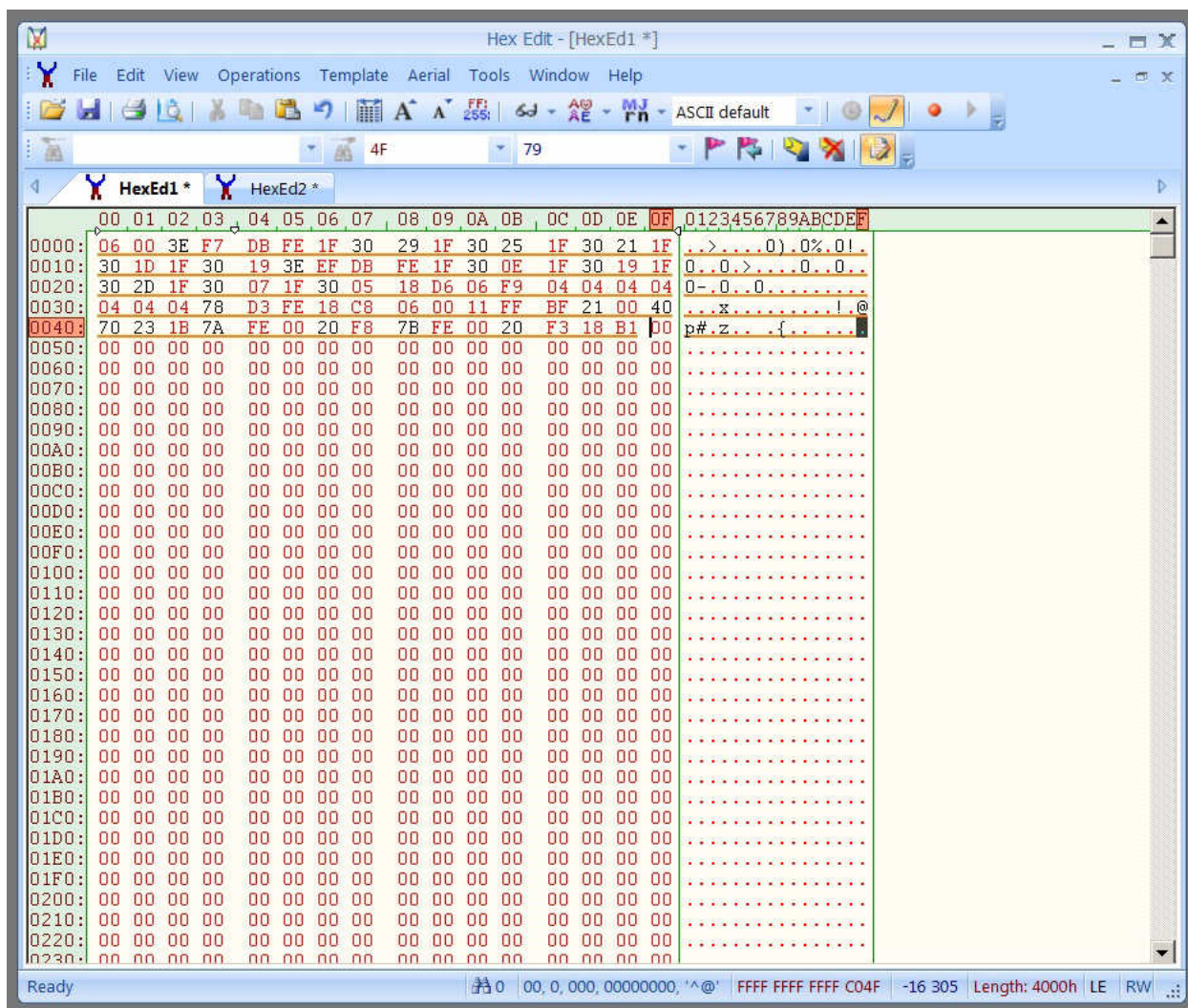


Рис. 3503. Редактор Hex Edit: Вставка шестнадцатичных данных с замещением.

Закрываем «HexEd2» (Close не сохраняя), а самый первый файл «HexEd1» записываем под именем «48.rom». Открываем «File» → «Save as...» и сохраняем его как «48.rom» Закрываем редактор и готовимся тестировать ПЗУ.

Перепишем «48.rom» в папку Spectasculator, и запускаем эмулятор, подключив только что созданное ПЗУ. На экране замигают квадратики:

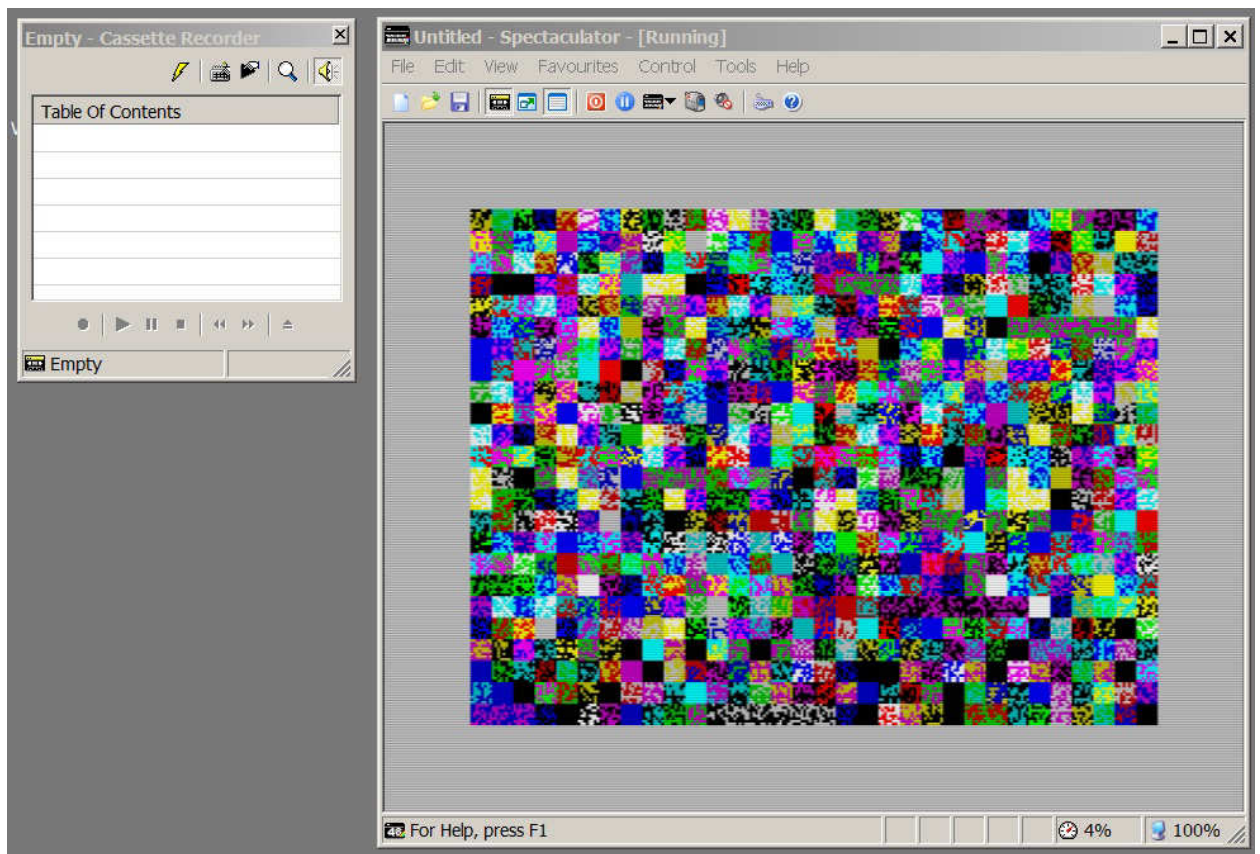


Рис. 3504. Проверка созданного ПЗУ на эмуляторе Spectaculator.

Нажмите клавишу 9, и запустится наша подпрограмма, которая очистит экран, и он станет черным. Это очистится все ОЗУ от мусора. Понажимайте клавиши от 1 до 7 и вы увидите, как от нажатых клавиш меняется цвет рамки. Наша программа полностью работоспособна. Нажав клавишу 8, программа выйдет на адрес 00079, в котором ничего нет, и пойдет гулять вниз по свободным адресам. Команда RET не имеет никакого смысла, так как возвращаться то некуда. Гуляя по памяти, компьютер снова вернется к выполнению программы с адреса 0. В этом можно убедиться, нажимая клавиши с цифрами. Снова от нажатия клавиш переключаются рамки:

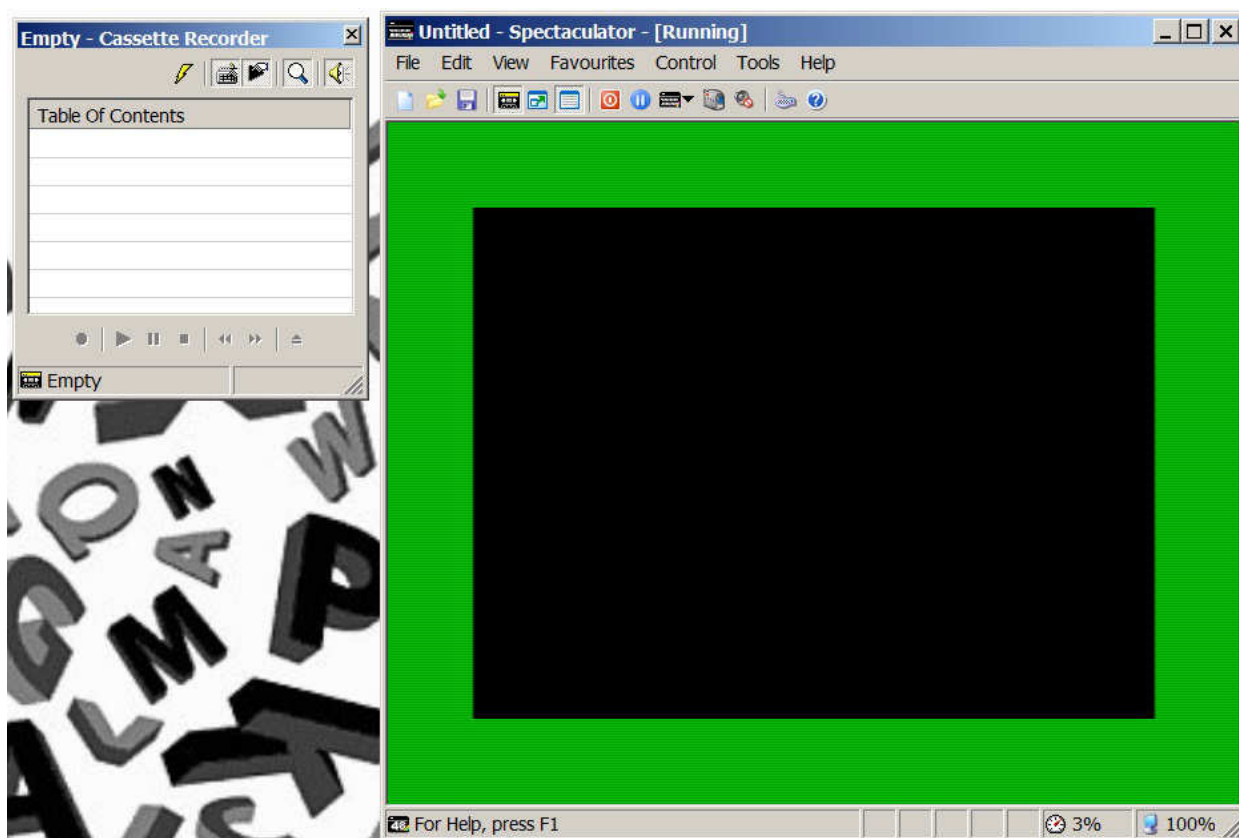


Рис. 3505. Spectaculator с созданным ПЗУ. Изменение цветов рамки от нажатия клавиши.

На самом деле, чтобы создать нормальную прошивку ПЗУ, следует также учитывать систему прерываний, и некоторые другие моменты, под которые изначально «затачивалась» стандартная программа ПЗУ.

ЧАСТЬ 4.

Создание блоков данных утилитами под windows XP и 7.

В этой части книги рассмотрим создание блоков данных с помощью программ под windows, без участия эмуляторов. Создание программ под Spectrum 48K сторонними программами дает еще больше возможностей и значительно облегчает работу.










Глава 1.

Создание и запись картинки с нижними строками.

Краткое содержание: утилиты ZX-Modules, основы работы в ZX-Paintbrush, создание картинки для ZX-Spectrum, прямое сохранение файла в *.tap / *.tzx формат.

Снова вернемся к сохранению картинки с нижними строками. Во 2-й части рассматривали вариант создания с помощью машинных процедур, напрямую обращаясь в подпрограмму записи из ПЗУ.

В этой главе будем рассматривать, как можно создать и записать картинку для спектрума программами из под windows. Можно воспользоваться прекрасным бесплатным пакетом программ, с официального сайта: <http://www.zxmodules.de/>. Пакет программ создан для редактирования и создания программ для спектрума прямо из под windows. Скан с сайта со списком программ, описанием и рекомендациями:

	Module name:	Description:	Latest version:
	ZX-Explorer	Opens or displays ZX-Spectrum emulator files on drives, also detects programs/games automatically. Can also be used as a thumbnail viewer. It manages also the display of compressed files.	July 2009
	ZX-Favourites	Stores most wanted programs with their game information in a database. Can import S.G.D. and other databases.	October 2008
	ZX-Preview	Shows screen\$, basic listings, system variables, etc. of ZX-Spectrum emulator files.	January 2013
	ZX-Blockeditor	Edits the blocks of ZX-Spectrum emulator files, e.g. all TZX format blocks. Create DSK or TRD disk image files. ZX-Blockeditor can open ZX-Editor and ZX-Paintbrush for Basic and SCREEN\$ block editing!	January 2013
	ZX-Editor	Edits ZED-files and many more ZX-Spectrum emulator files directly. Contains ZX-BASIC and BETABASIC syntax checker and syntax help. Lets you run your edited file.	January 2013
	ZX-Paintbrush	Graphic editor. Edits Spectrum screen\$ and other picture formats. Also Multi-block editing for tape and disk image files is supported. Lets you run your edited file.	January 2013
	ZX-Central	Manages your favourite ZX-Spectrum programs (emulators and tools); binds any Windows-compatible ZX-Spectrum program into a panel; lets you design layouts; helps you updating your programs and tools	coming in 2013
	ZX-Gamestatistics	Completes scanning Spectrum archives and building statistics about similarities of the games	planned for 2013
	ZX-Assembler	A complete Assembler/Disassembler system	coming in 2013

Module connection:

I recommend to download all of my programs, especially ZX-Blockeditor, ZX-Paintbrush and ZX-Editor communicate with each other.

Most of the ZX-Modules applications can receive commands from other modules. Some modules (e.g. ZX-Editor and ZX-Paintbrush) can edit data blocks inside embedded dialogs as well.

Рис. 4100. Сайт ZX-Modules. Скан на 31 января 2013 года.

Как видно из картинки, много интересного вышло, но немало интересного еще и готовится к выпуску. Автор советует установить весь пакет программ, которые могут взаимодействовать друг с другом. По нынешним меркам программы небольшие, и устанавливаются быстро. Программы прекрасные, но на первый взгляд, довольно сложные.

Сейчас попробуем нарисовать тестовую картинку в программе ZX-Paintbrush, с нижними строками и сохранить ее в формате магнитной ленты в область экрана. В опыте я буду использовать версию 2.2.9.1 конца 2012 года, но пока я работал над этой главой, уже вышла новая версия 2.2.9.6 от января 2013 года.

Откройте ZX-Paintbrush. Нажмите кнопку «File» в левом углу окна, и в выпадающем меню нажмите «New», или комбинацию **CTRL+N** напрямую из окна редактора. Откроется окно выбора форматов записи будущей картинки:

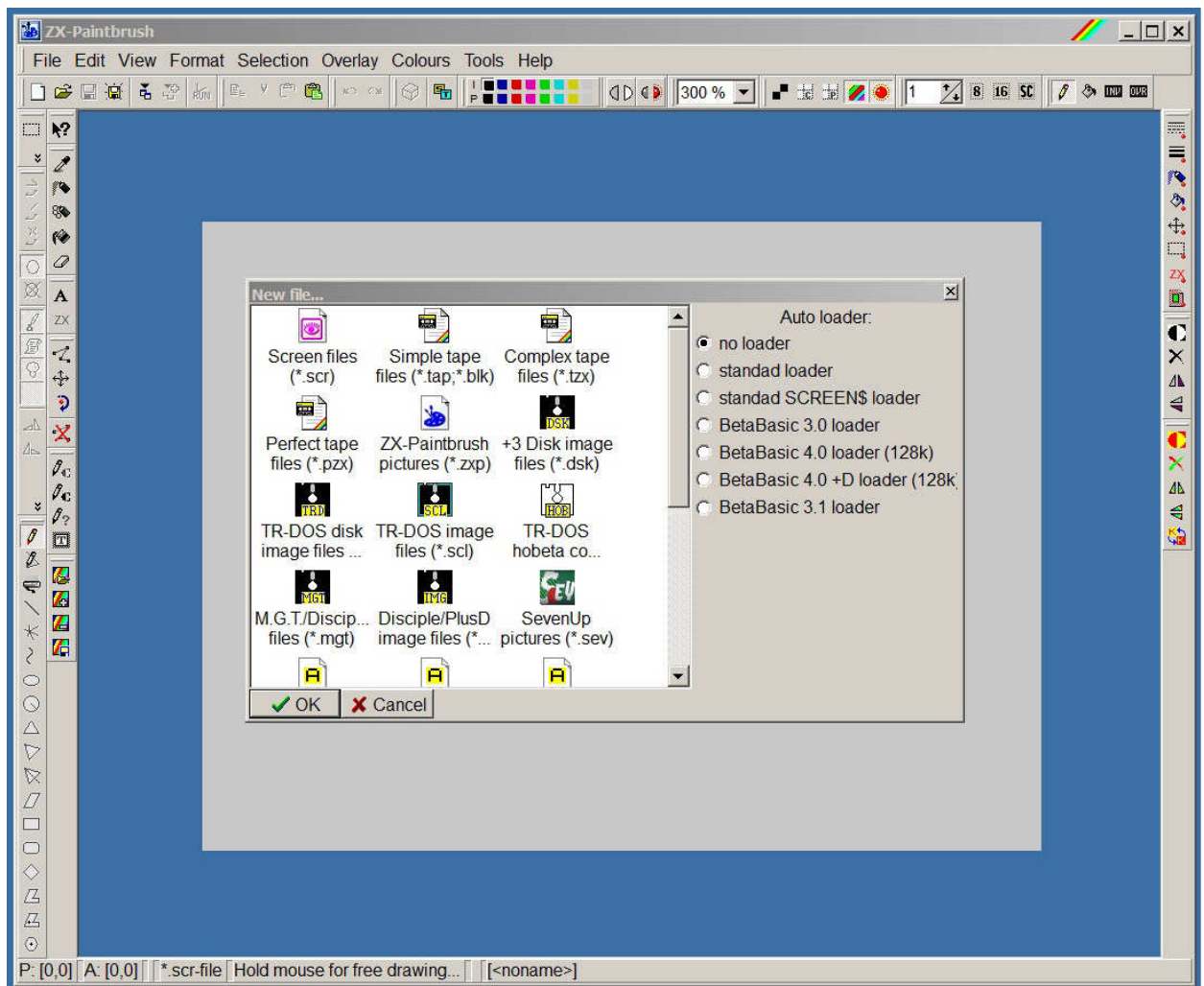


Рис. 4101. ZX-Paintbrush. Окно выбора формата для создания нового файла.

Выберем самое первое и самое простое: «*Simple tape files*» и нажмем на него. В правом углу высветится дополнительное меню, предлагающее набор готовых шаблонов для компоновки будущего файла. Некоторые шаблоны предполагают доработку в программе ZX-Editor, и пока рассматриваться не будут.

Выберем самый простой шаблон «*no loader*», что будет означать одиночный блок «*Bytes :*» картинки, загружаемый по команде `LOAD ""CODE`. Нажмем «OK». Откроется окно «*Add screen header*» где в строке заголовка попросят ввести имя будущему заголовку «*Bytes :*» Дополнительно нажав на кнопочку с карандашиком внизу окна, справа откроется дополнительная секция окошка «*Memory Editor*»:

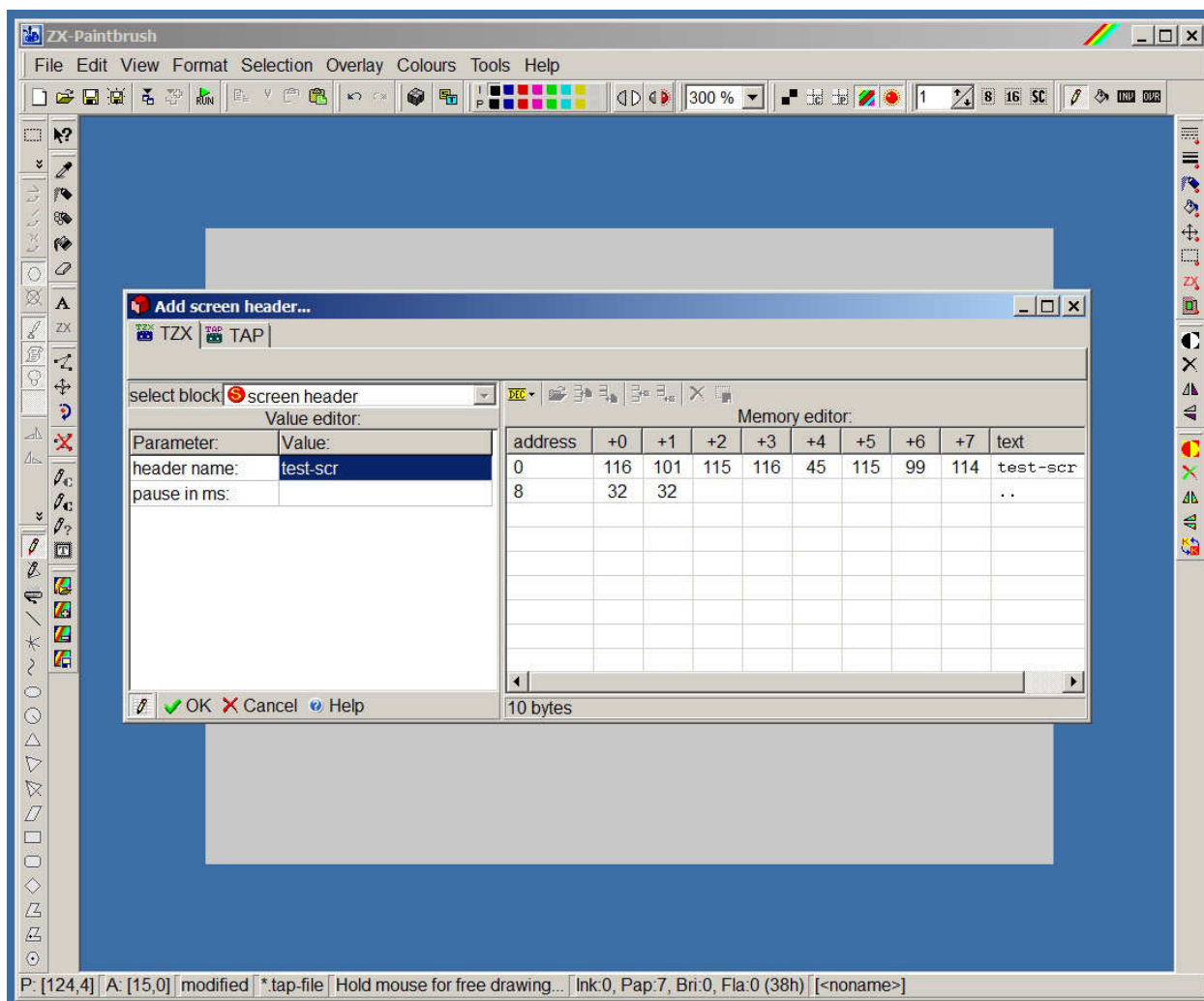


Рис. 4102. ZX-Paintbrush. Секции Value editor и Memory editor.

Нажав «OK» выйдем в редактор для создания рисунка. Редактор имеет очень много функций для редактирования, вставки текста и даже редактирования шрифтов, а также имеет подробное описание со справкой, но только на английском. Русские шрифты также не поддерживаются.

Попробуем создать какой-нибудь произвольный рисунок, чтобы он захватывал нижние строки:

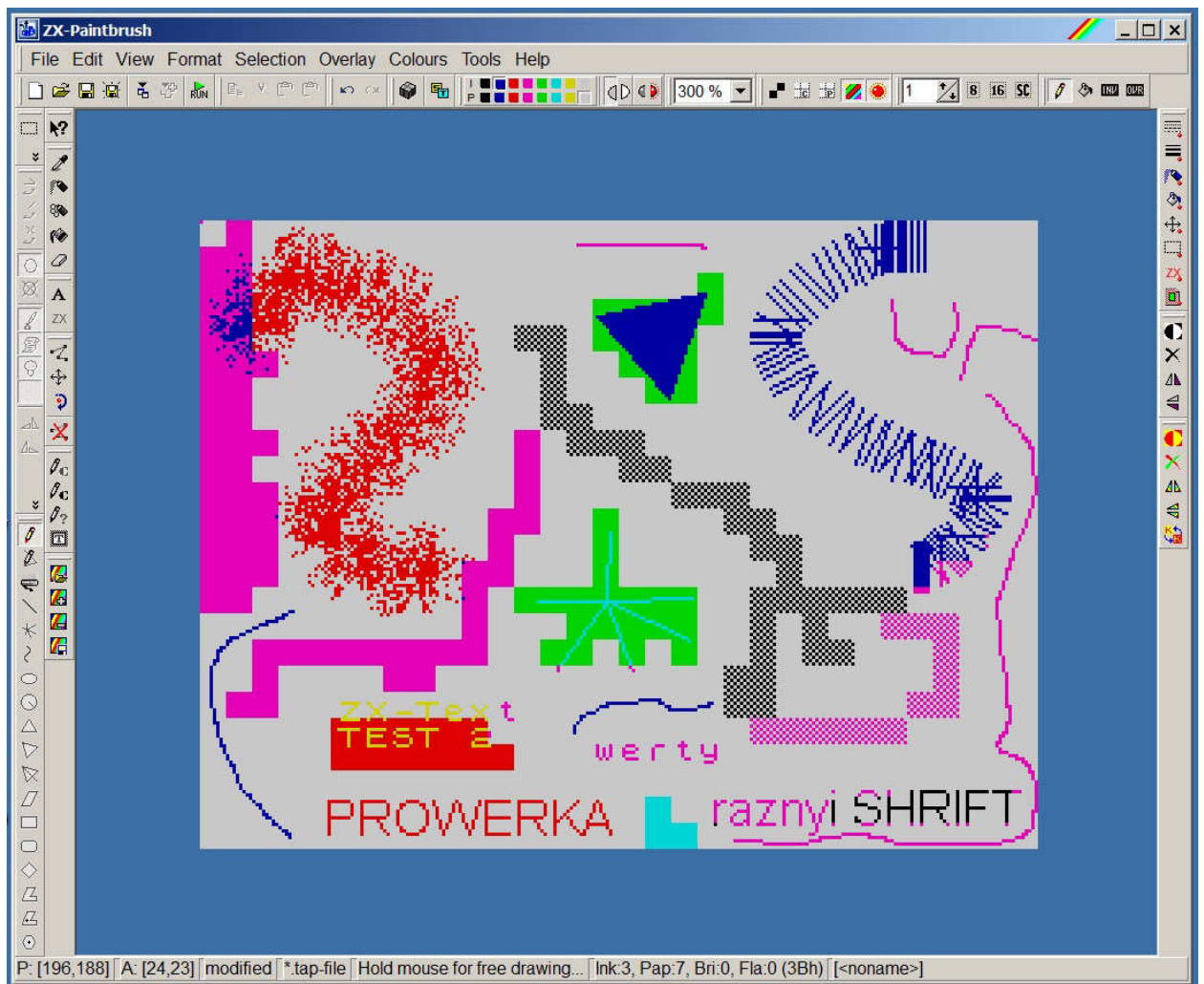


Рис. 4103. Редактор ZX-Paintbrush. Создание тестовой картинки для записи.

После того как вы закончите рисунок, в левом верхнем углу редактора, снова нажмите кнопку «File», а в меню выберите «Save». Откроется окно сохранения файла. Введите имя сохраняемому *.tap файлу (для PC) например также «test-scr.tap»:

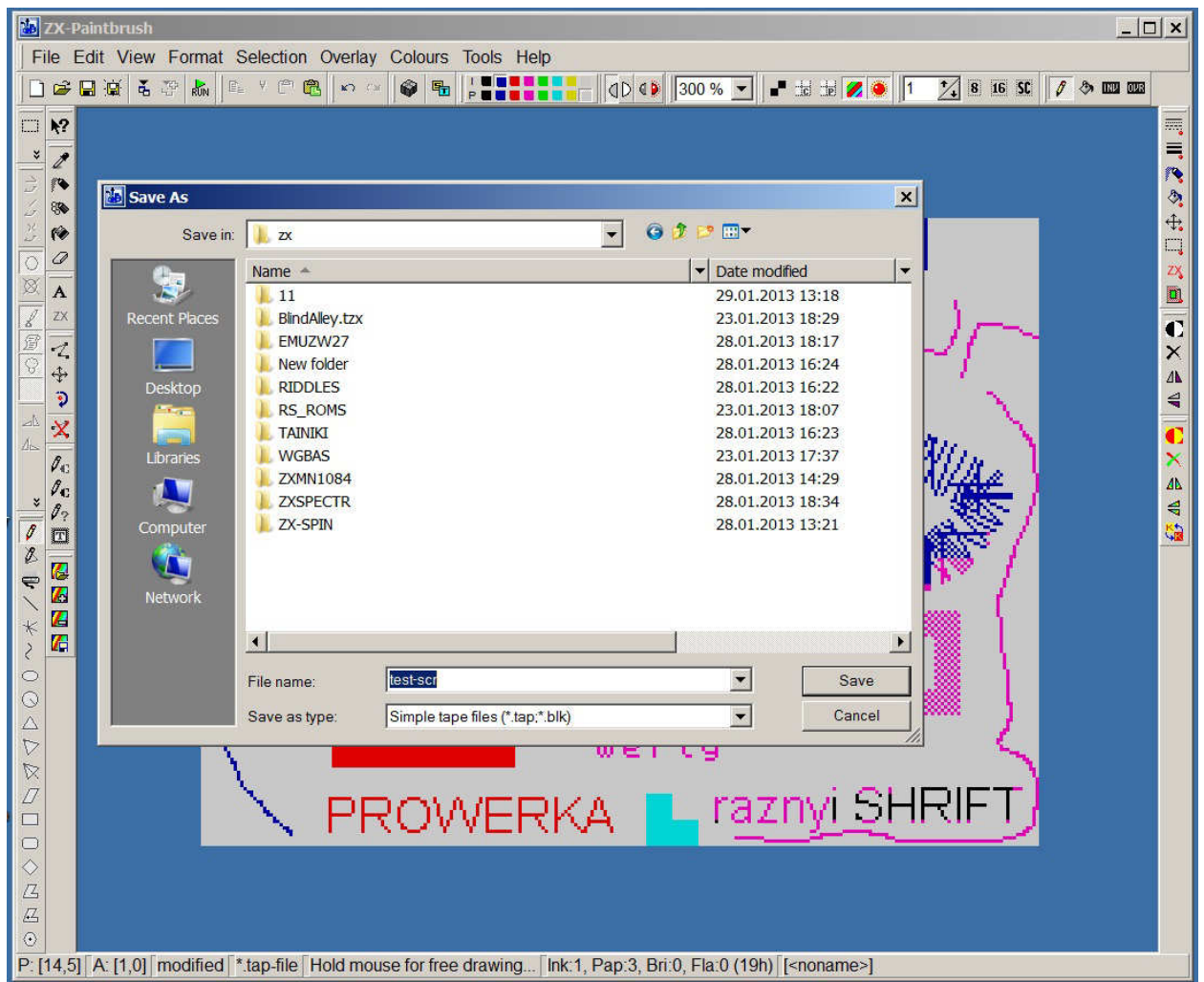


Рис. 4104. Редактор ZX-Paintbrush. Окончательное сохранение файла.

Нажмите «Save» и ваш рисунок сохранится в *.tap формате. Закройте ZX-Paintbrush, нажав «File», а в нем «Exit». Итак, наш «синтетический» *.tap файл с картинкой, выполненный программами под windows, без участия эмуляторов, выполнен. Осталось только его проверить на эмуляторе. Откройте созданный файл «test-scr.tap» в Spectaculator. Для загрузки введите строку `LOAD ""CODE: PAUSE 0,` и нажмите ENTER:

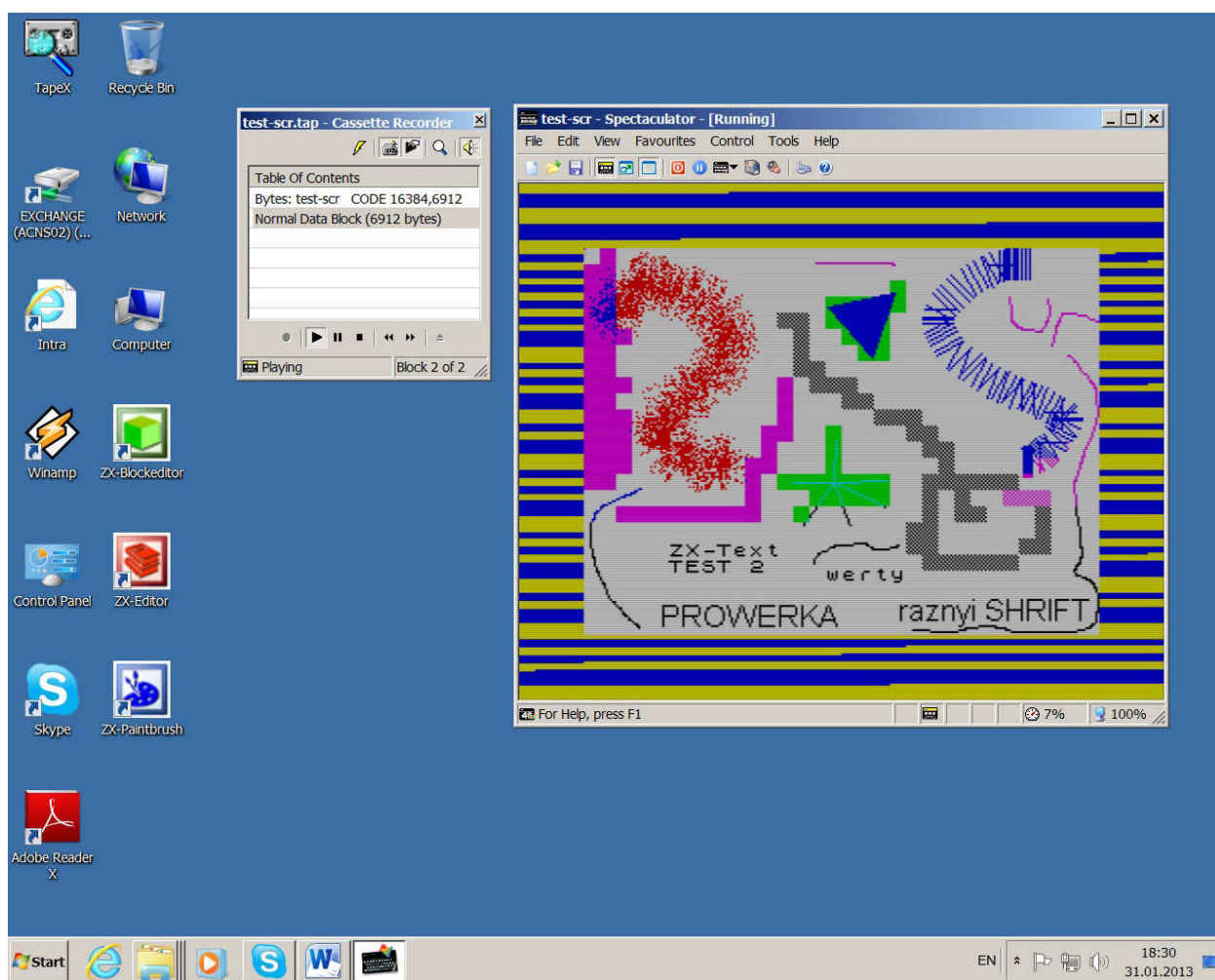


Рис. 4105. Загрузка картинку на эмуляторе Spectaculator в процессе «перекраски».

Оператор **PAUSE** ☐ нужен для того, чтобы после загрузки картинку, компьютер остановился ожидая нажатия любой клавиши, и не затер рисунок в двух нижних служебных строках. Полюбовавшись на картинку можно нажать клавишу, и нижние строки затрутсся, выдав сообщение:

☐ OK , ☐ : 2

Итак, как мы убедились, создавать блоки данных для спектрума можно и без помощи самого эмулятора.

Глава 2. Создание простейшей BASIC программы без эмулятора.

Краткое содержание: основы работы в ZX-Editor, работа с заголовком программы, создание BASIC программы без эмулятора.

В предыдущей главе мы рассмотрели создание блоков с графикой без помощи эмулятора, и сделали тестовую картинку, кратко ознакомившись с некоторыми функциями программы ZX-Paintbrush. Теперь попробуем создать простейший загрузчик на BASIC для этой картинку, также без помощи эмулятора. Для этих целей можно воспользоваться программой ZX-Editor. В опытах я буду использовать версию 2.2, вышедшую в январе 2013 года.

Откройте ZX-Editor. В левом верхнем углу окна редактора мигает курсор. Так как мы будем делать вспомогательный загрузчик на BASIC, то первым делом на панели

программы нажмите кнопку с пиктограммой «48k» (*displays tokens at 48k tokens*). Выберите оптимальный масштаб размера символов в редакторе (*zoom factor*), например 200%. Удобно расположите и растяните окно на рабочем столе, и можно приступать к работе. Цвет пустого пространства внутри окна, редактор заимствует с рабочего стола в windows.

Попробуем создать простейший загрузчик на BASIC и записать одиночным блоком в формате *.tzh. Структура меню очень похожа на рассматриваемый в предыдущей главе ZX-Paintbrush. В редакторе нажмите на кнопочку «File». Откроется выпадающее меню «New file...». Как видно из меню, кроме всевозможных кассетных и дисковых форматов записи, сохранить свою BASIC программу можно и в обычном файле *.txt. На этот раз выберем «Complex tape files (*.tzh)», и слева появится целых 9 шаблонов записи программы. Снова оставим самый простой «no loader»:

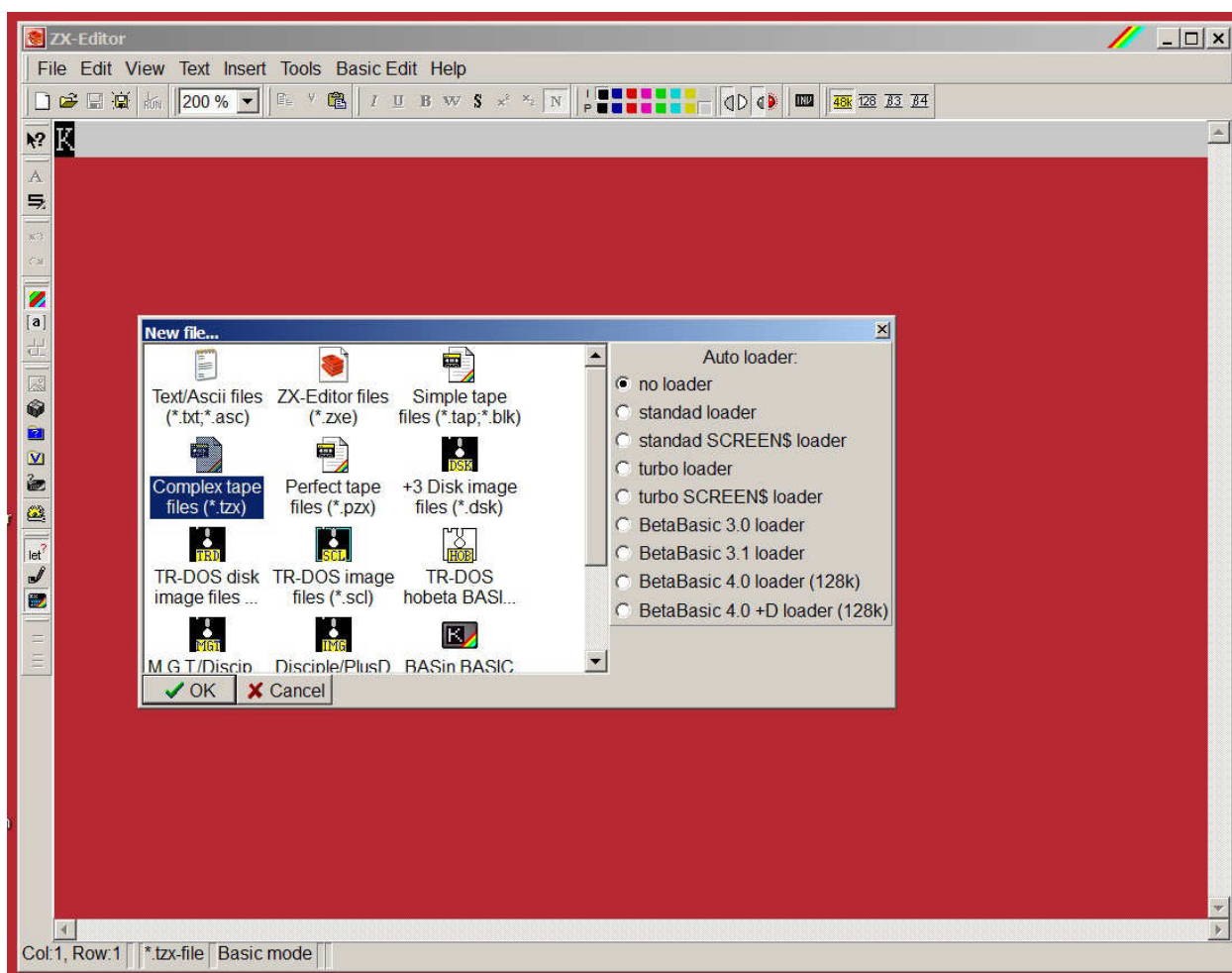


Рис. 4200. Редактор ZX-Editor. Выбор шаблонов с загрузчиками для записи блоков данных.

Нажмем «OK». Это окно закроется, и откроется окно редактирования заголовка (*Add program header...*). В верхней строке «Select block», по умолчанию стоит «Program header». Но нам нужно, чтобы после загрузки программы, автоматически загружалась картинка. Выберите из выпадающего меню «Program autostart header». После выбора этой опции, нижняя табличка расширится до 5 строк. В колонке Parameter появится дополнительная строка №4 «autostart line:» в которой по умолчанию будет записано значение «0». Тут нужно указать номер строки, с которой программа автоматически запустится после загрузки. Можно оставить по умолчанию, если собираетесь делать строку 0. Этот параметр будет являться аналогом оператора LINE, в строке SAVE " " LINE 0, для записи блока. В строке №1 «program name:» латинскими буквами

введите имя будущему BASIC блоку. Например «TEST-SCR» и введите его. Остальные параметры пока не трогайте, они выставятся автоматически после окончательной записи:

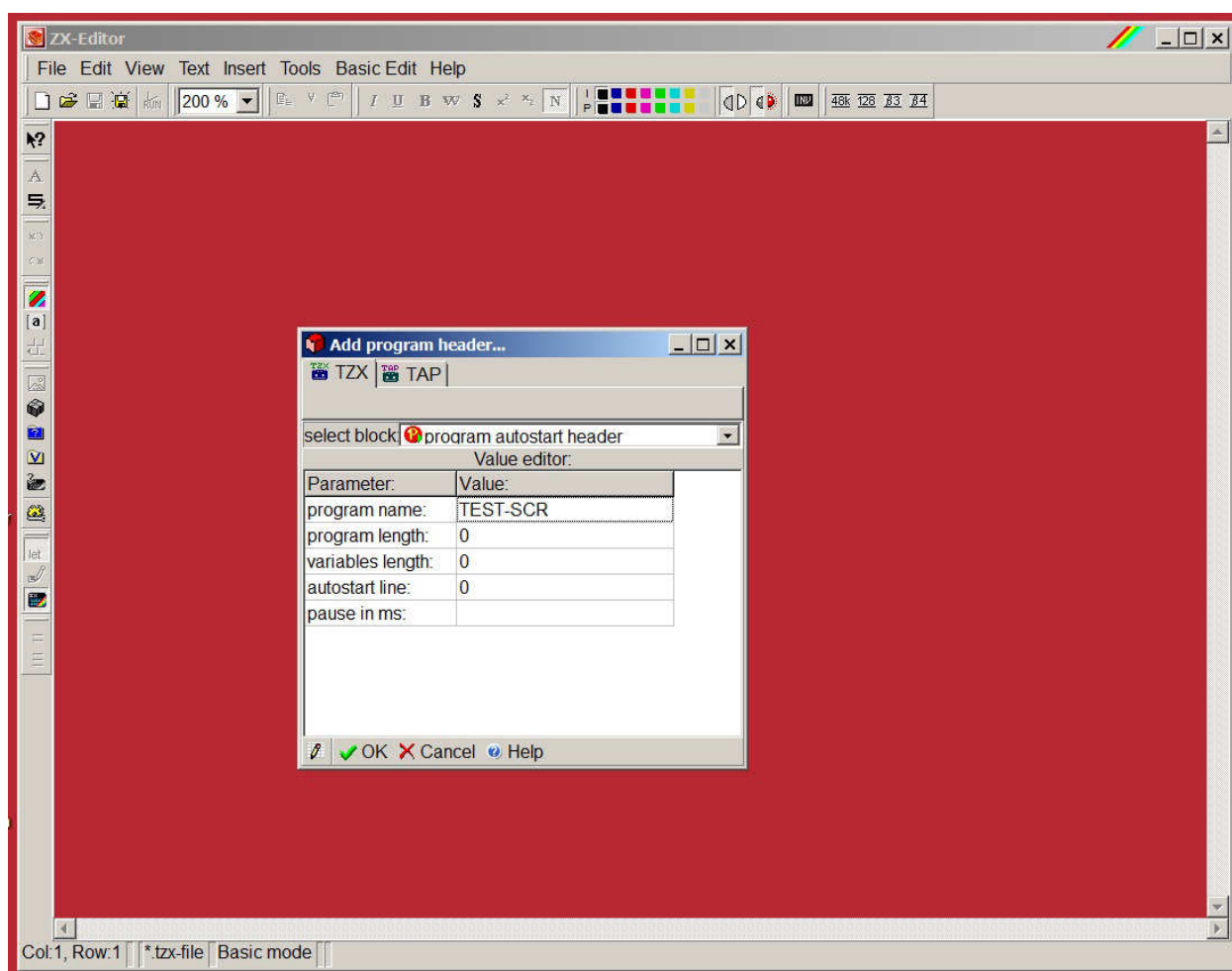


Рис. 4201. Редактор ZX-Editor. Создание и редактирование параметров заголовка блока «Program:».

В левом нижнем углу, также как и в ZX-Paintbrush, можно открыть дополнительную секцию окна «Memory editor», в которой отобразится структура заголовка. Нажимаем «OK» и выходим в редактор для создания программы. Для удобства, в режиме 48K, все команды набираются не побуквенно, а точно также как в спектруме. Абсолютно также меняются и курсоры. Редактор является гибридом блокнота и спектрума. Он содержит в себе интерпретатор настоящего спектрумского бэйсика, и можно писать полноценные программы, только гораздо удобнее. Мышкой можно ставить курсор в любую часть строки и редактировать любой символ или оператор. В редакторе также снято ограничение на нулевую строку. Попробуем набрать программу:

```
0 BORDER 5: PAPER 5: CLS
1 LOAD ""CODE
2 PAUSE 0
```

В редакторе выглядеть это будет следующим образом:

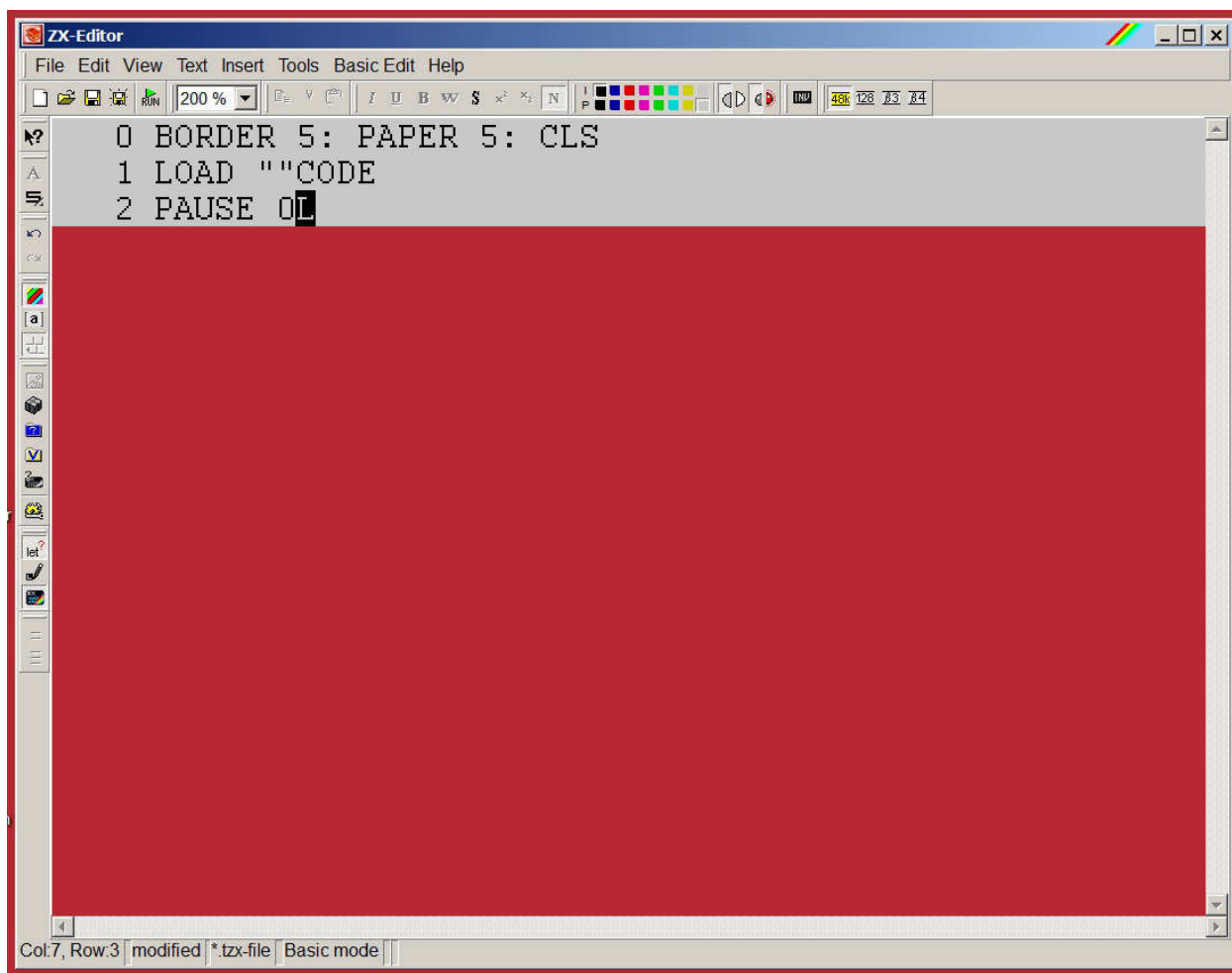


Рис. 4202. Редактор ZX-Editor. Создание программы на BASIC в режиме 48K.

Программа готова. Теперь сохраним ее окончательно. Нажмем кнопку «*File*», а в открывшейся вкладке «*Save file*» выберем папку, в которую будем записывать, и зададим имя файлу, например «*scr-loader*»:

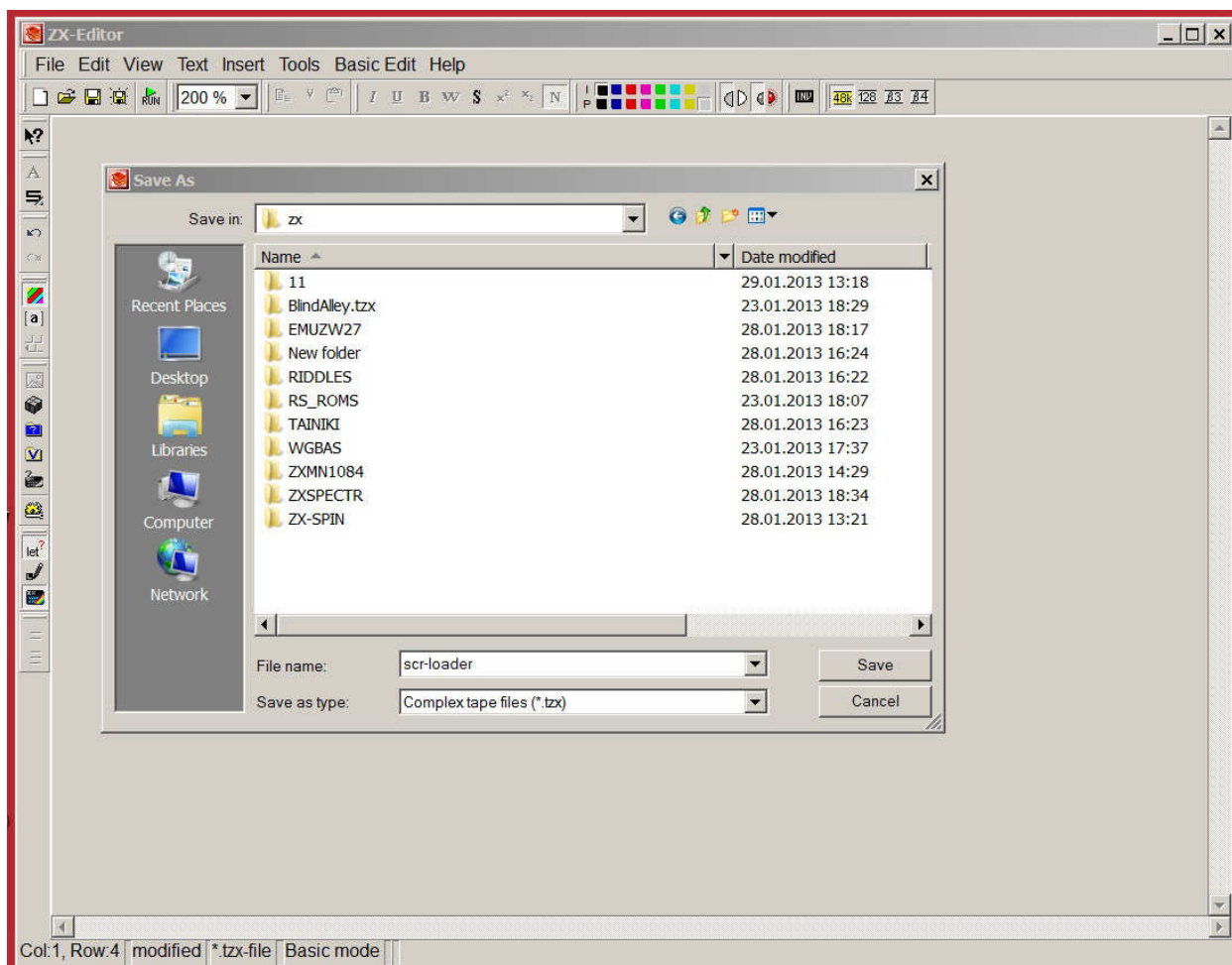


Рис. 4203. Редактор ZX-Editor. Сохранение BASIC программы в *.tzx с автозапуском.

Нажимаем «Save», окно закрывается, и программа записана. В редакторе, слева от масштаба, есть кнопка «RUN». Если у вас Spectaculator настроен в качестве основного эмулятора, и расширения файлов спектрума закреплены за ним, то при нажатии откроется Spectaculator и запустится программа.

Выходим из редактора, нажав «File», и в самом низу выпадающего меню «Exit». Теперь проверим созданную «синтетическую» BASIC-программу, загрузив ее в эмуляторе. После загрузки программы экран станет голубым, и компьютер будет ожидать блока «Bytes : » Это будет говорить о том, что программа заработала правильно. Теперь нажмите BREAK, а затем ENTER. На экране эмулятора вы увидите текст нашей BASIC программы:

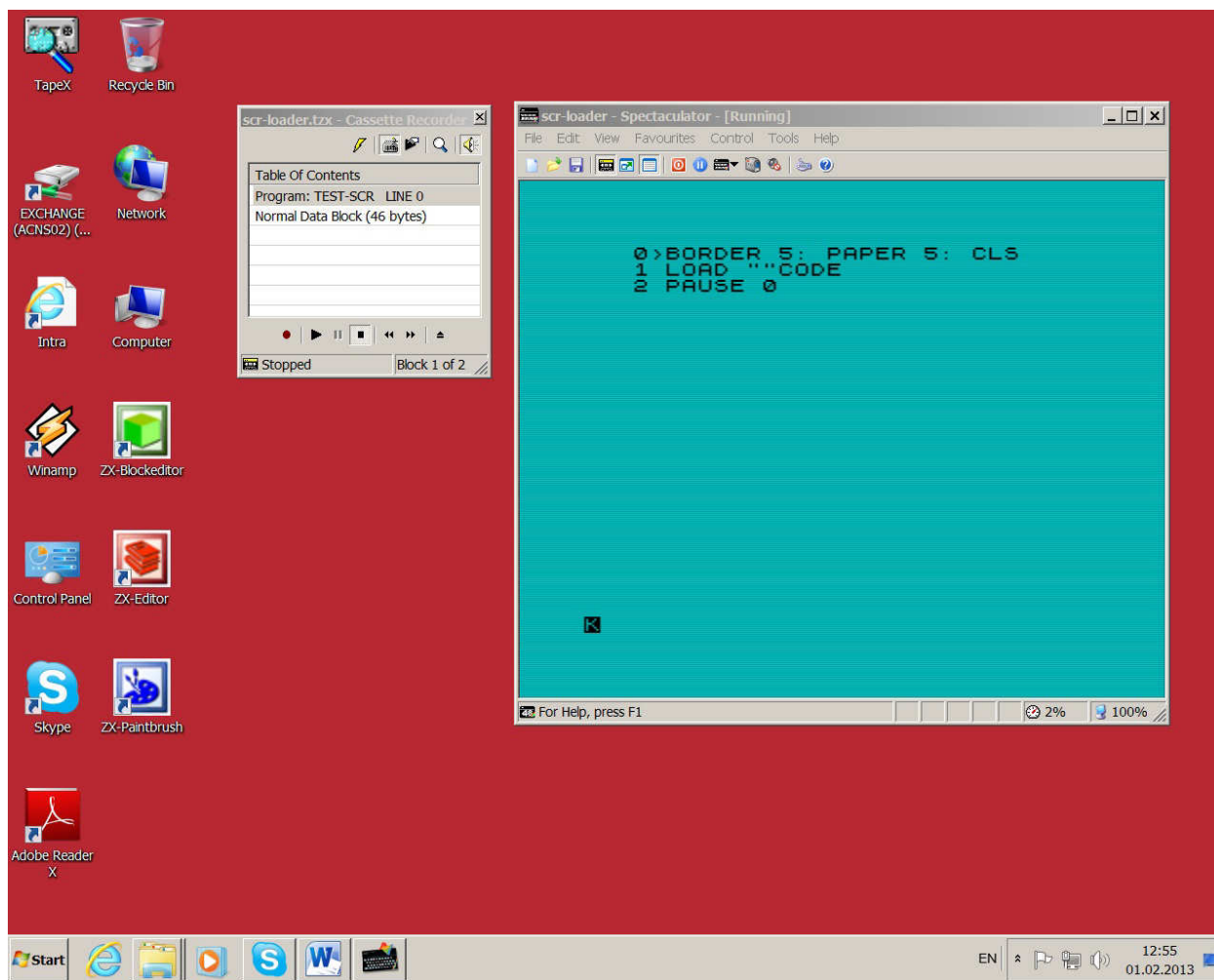


Рис. 4204. Проверка искусственной BASIC-программы на эмуляторе Spectaculator.

Итак, мы только что создали еще одну «искусственную» программу для спектрума. Теперь осталось склеить обе программы для полноценной работы. Об этом в следующей главе.

Глава 3. Основы работы с блоками и заголовками магнитофонных файлов.

Краткое содержание: знакомство и основы работы с Tapir и ZX-Blockeditor, добавление и перемещение блоков данных, формирование новых блоков данных, установка зазоров между блоками.

В предыдущих главах создали блок кодов «**Bytes** : » с картинкой, и программу-загрузчик на бейсике для нее, без помощи эмуляторов. Теперь настало время собрать написанные программы в одну, и рассмотреть основы работы и редактирования блоков данных таким же образом. Для этих целей под windows имеются несколько программ. Рассмотрю две, наиболее удобные, которые мне попались. Первая это Tapir v1.0, которая предназначена только для работы с файлами образа магнитной ленты. Поэтому она наиболее простая. Другая программа ZX-Blockeditor, из рассмотренного ранее, пакета «ZX-Modules». Она редактирует не только магнитофонные файлы, но и дисковые.

Сначала рассмотрим, как работать с блоками данных с помощью программы Tapir. Эта программа не требует установки, и открывается прямо из папки, в которую скопирована. Для удобства, папку с программой, лучше скопировать в корневой каталог

диска. После запуска программы появится окно, разделенное на две части, а в верхней панели кнопки «*Left*», «*Right*», «*Block*» и «*Help*».

Перетащите левой кнопкой мыши в окошко «*Left*», созданный двумя главами назад, блок с картинкой «*test-scr.tap*». В окне отобразятся два последовательно расположенных блока. Первый будет заголовок, второй блок с байтами данных. Нажмите мышкой на один из них, и он выделится черным цветом. Внизу, под окошками, появится информация о структуре и сигнале этого блока:

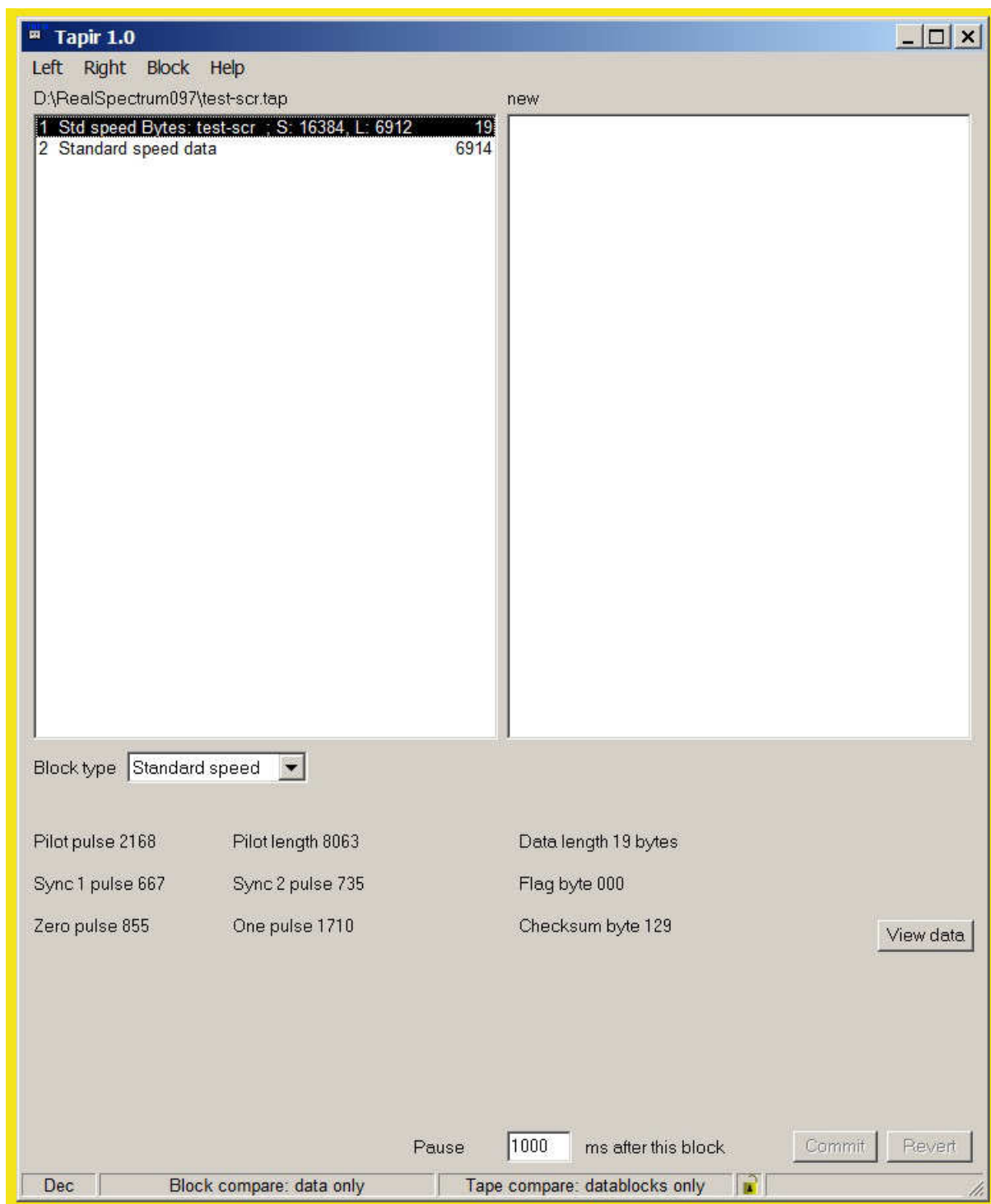


Рис. 4300. Программа Tapir. Окно с редактируемым блоком данных «*Bytes*:».

В самом низу окна находится беленькое окошечко, с числом 1000: «*Pause [1000] ms after this block*». Изменяя значение времени в миллисекундах, можно изменить паузу между заголовком, и следующим за ним сигналом блока данных. Убрав

промежутков, блоки заголовка и данных будут звучать без паузы. Подтверждение измененной величины нужно принять, нажав кнопку «Commit».

Аналогичный прием загрузки тестовой программы без промежутков был применен на кассете, прилагаемой к компьютеру «Дубна 48K», приборостроительного завода «ТЕНЗОР» (Московская область).

Под левым окном программы есть еще одна функция «*Block type*», в окне которого по умолчанию выбрано «*Standard speed*». В открывающемся меню есть несколько шаблонов типов формирования звукового сигнала, обеспечивающего загрузку программы с магнитной ленты.

«*Standard Speed*» - готовый шаблон со стандартным сигналом загрузки, воспринимаемый командой `LOAD`, или стандартной процедурой считывания из ПЗУ. Оставим этот шаблон.

При выборе «*Turbo speed*» цифры, справа от параметров, станут доступны для редактирования. Зная параметры сигнала турбозагрузки, вы можете сами выставить параметры формируемого сигнала. Но для считывания такого сигнала придется создавать собственный загрузчик, который можно разместить как в ОЗУ, так и в ПЗУ эмулятора, модернизировав файл виртуального ПЗУ (файл *48.rom*).

Опция «*Pure data*», это чистые данные без пилот-тона, и «*Generalized data*», где вам также вручную предлагается выставить параметры формируемого сигнала записи.

Кнопочкой «*View Data*» можно посмотреть данные выделенного блока. Выделите мышкой «*Standard speed data*» и нажмите эту кнопку. Откроется новое окно «*Data Window*» и вы увидите следующее:

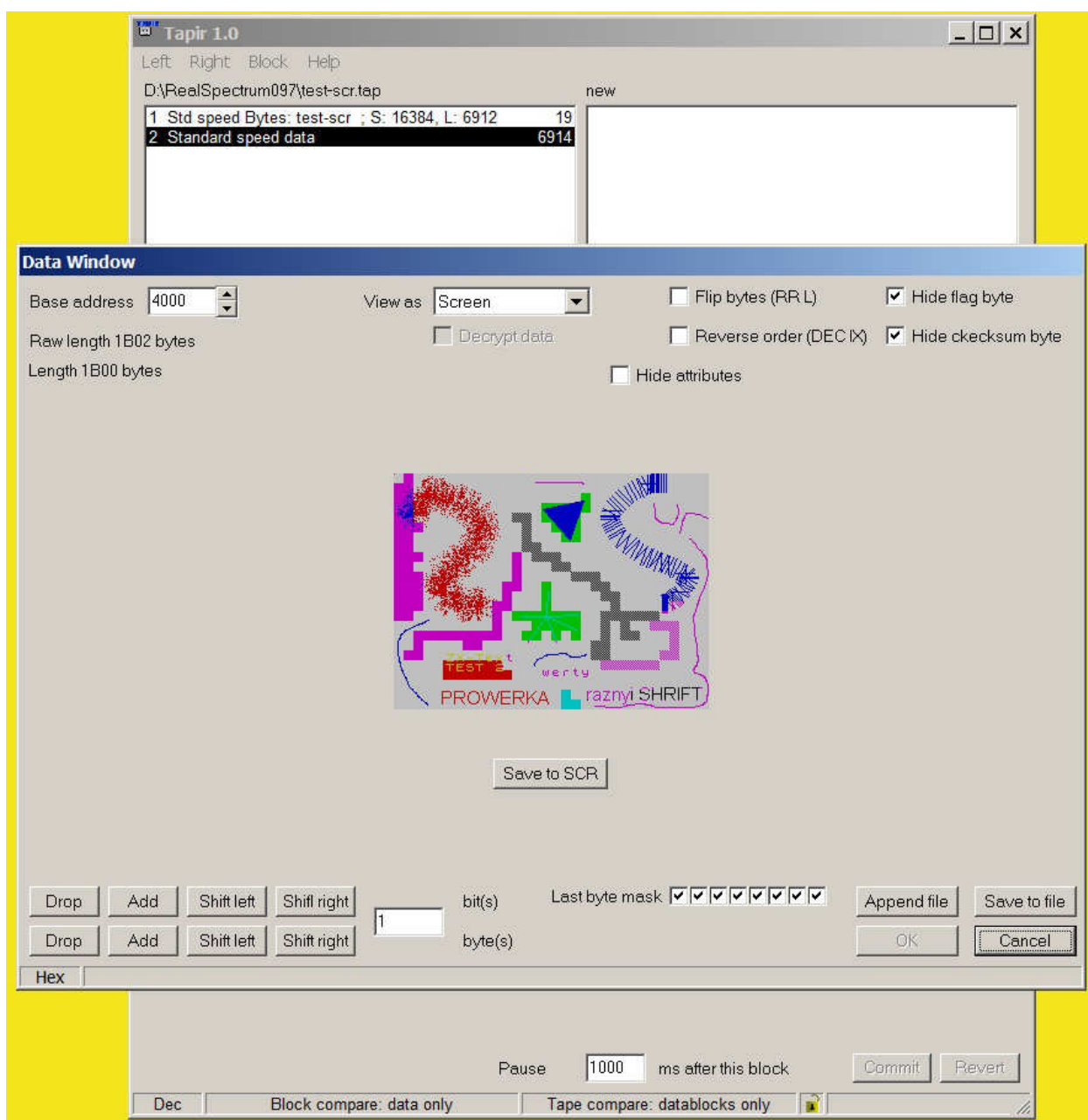


Рис. 4301. Программа Tapir. Окно просмотра и редактирования данных «Data Window».

В выпадающем списке окна «View as» блок по умолчанию определился как «Screen» (картинка). Его также можно просмотреть в режиме «Dump» (байты данных), «BASIC listing» (Если это блок бэйсик программы), «Disassembly» если это программа в машинном коде, или «System variables».

Причем дизассемблировать можно с любого адреса, на котором стоит курсор в режиме «Dump». Дизассемблированную программу можно скопировать в буфер обмена и перенести в любой текстовый редактор или ассемблер эмулятора.

По желанию можно изменить и стартовый адрес блока (*Base address*). Сейчас мы только ознакомились с некоторыми функциями и ничего менять не будем. В правом нижнем углу окна нажмем «Cancel», и снова выйдем в основную программу.

Теперь добавим блок загрузчик на бейсике, который создали в предыдущей главе и записали под именем «scr-loader.tzx» Для этого нажмем кнопку «Left», а в выпадающем меню «Append file...». В открывшемся окне выберем «scr-loader.tzx»:

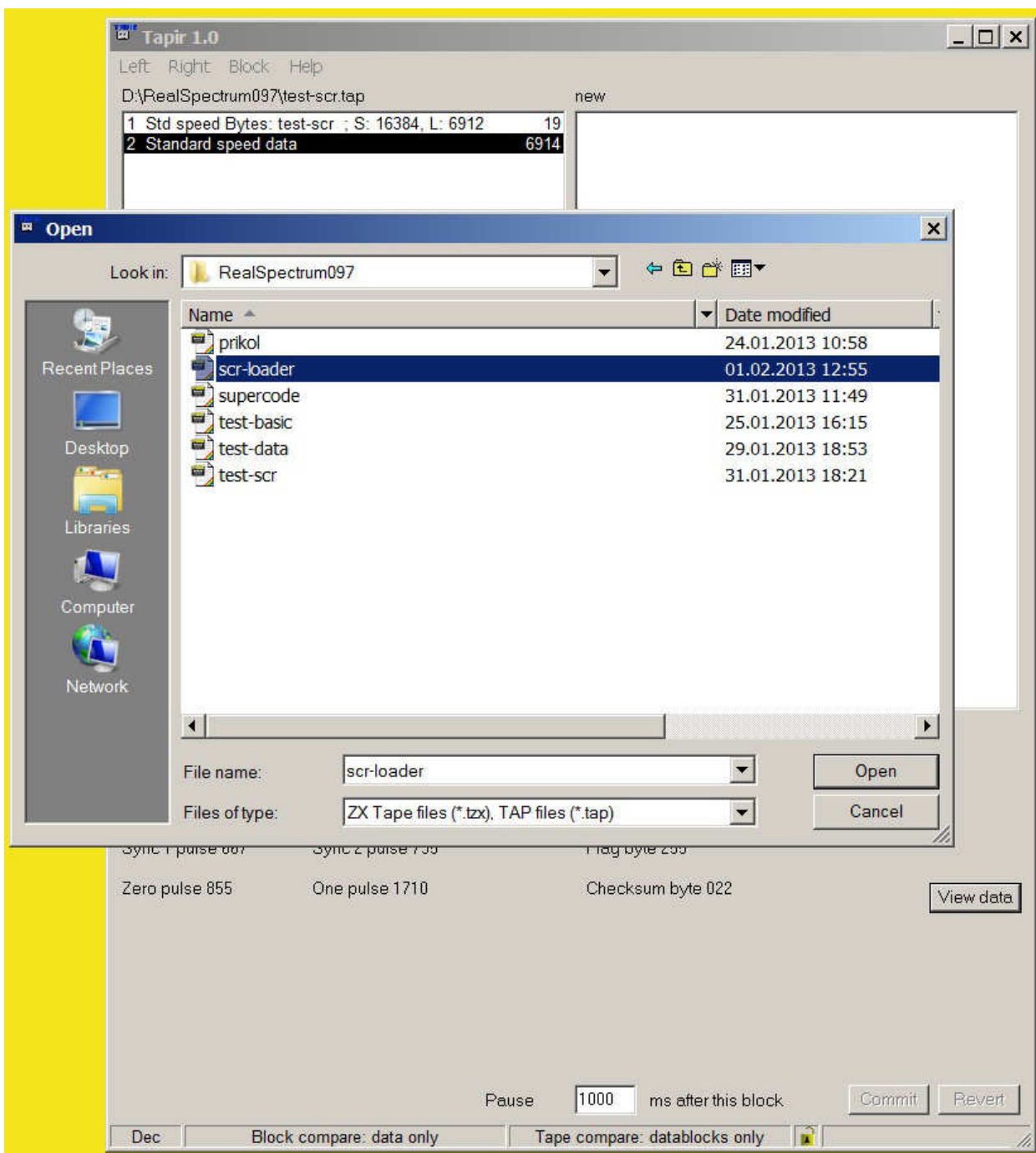


Рис. 4302. Программа Tapir Добавление блока к существующему файлу и их склеивание.

Нажимаем «Open», и наш блок на бейсике добавляется вниз, под пунктами 3 и 4, не стирая блоки картинки. Теперь нужно поменять местами блоки, потому что если так и записать, то вначале будет загружаться «Bytes:», а затем «Program:».

Чтобы это сделать, выделите мышкой блок заголовка программы из строки 3 «Std speed Prog: TEST-SCR : L: 46. P: 46. S: 0.», протащите его вверх списка, и отпустите. Теперь он встанет вверх списка под номером 1. Теперь вытяните блок данных этой программы «Standard speed data» длиной 48 байт, стоящий в конце списка, и поставьте его под заголовок программы в строку 2. Получится следующее:

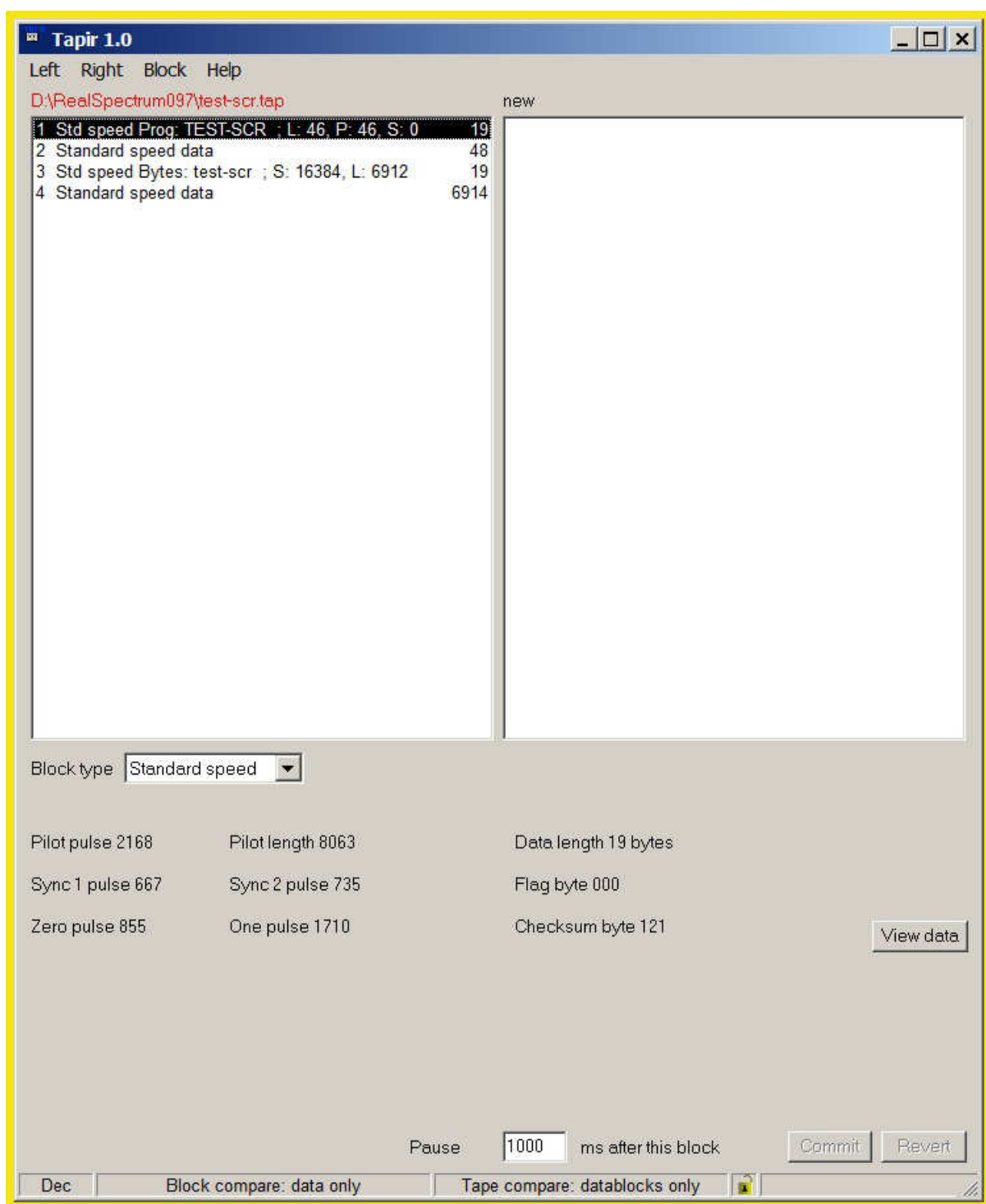


Рис. 4303. Программа Tapir. Формирование блоков данных в нужной последовательности.

Блоки будущего файла составлены в нужной последовательности. Но теперь посмотрите на данные программы, выделив блок в строке 2. Между блоками отсутствует пауза. Впишите внизу в белое окошко вместо «0», например стандартное значение «1000» и нажмите «Commit».

Теперь можно записывать полученный 2-х блоковый файл в формат *.tzh. Кстати, блоки можно между собой не только перетасовывать, но и удалять (клавиша DEL), или создавать разные специфические вручную.

Нажмите кнопку «Left», а в меню «Save as...». Запишите файл под именем «my-screen.tzh». Расширение указать обязательно, потому как файл, так и запишется без него, если в названии не поставить. Это недоработка программы.

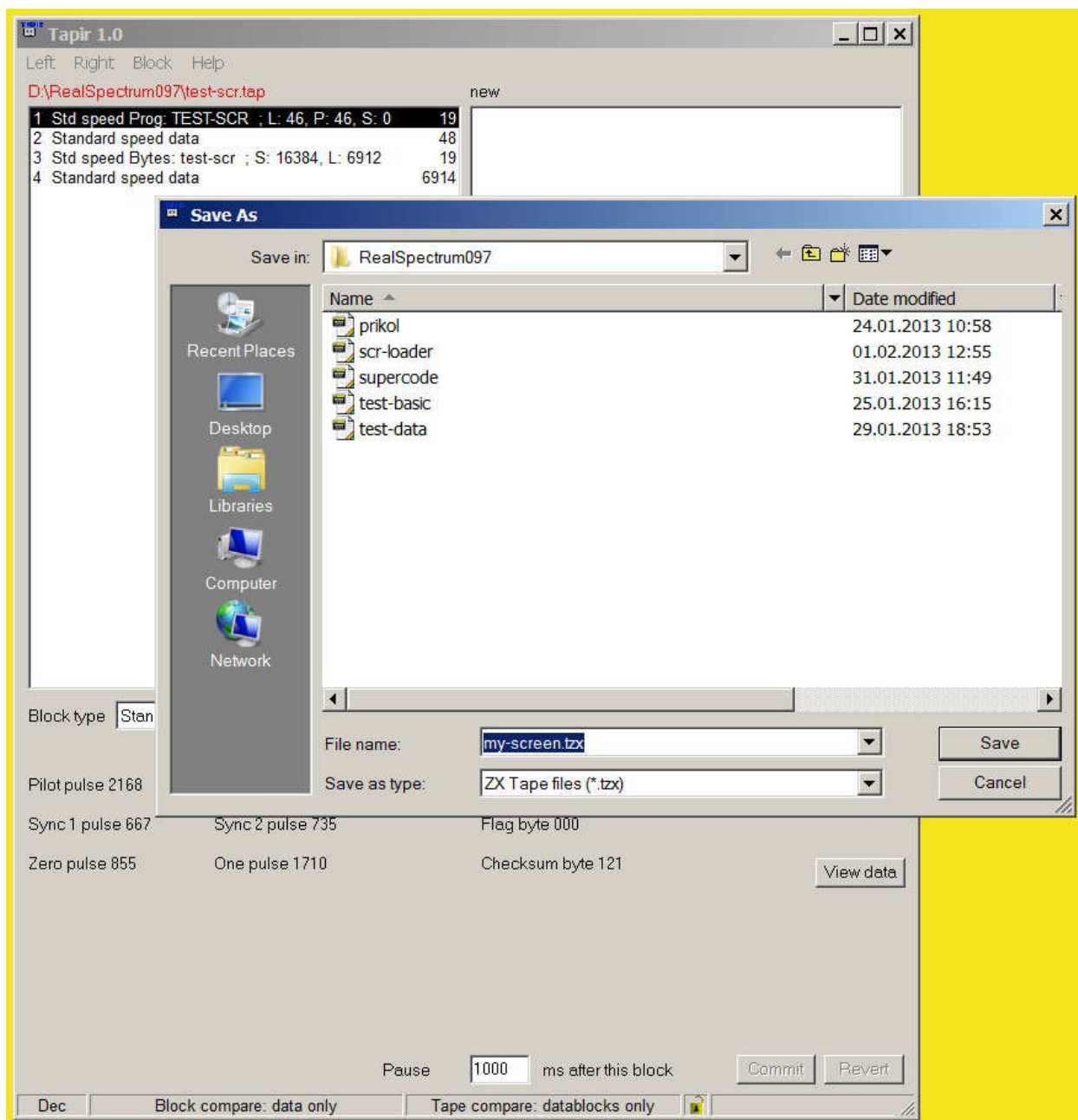


Рис. 4304. Программа Tapir. Сохранение .tzx файла, созданного из двух файлов разных форматов.

Можно прослушать созданный файл прямо из программы. Для этого, после нажатия «Left», нужно выбрать меню «Play». Откроется портативный плеер и зазвучит созданный файл.

Теперь приступим к самому главному – проверке работоспособности слепого файла *.tzx. Откроем его в Spectaculator и загрузим программу по `LOAD ""ENTER`. Сначала загрузится BASIC программа, экран станет голубым, и начнется загрузка следующего блока:

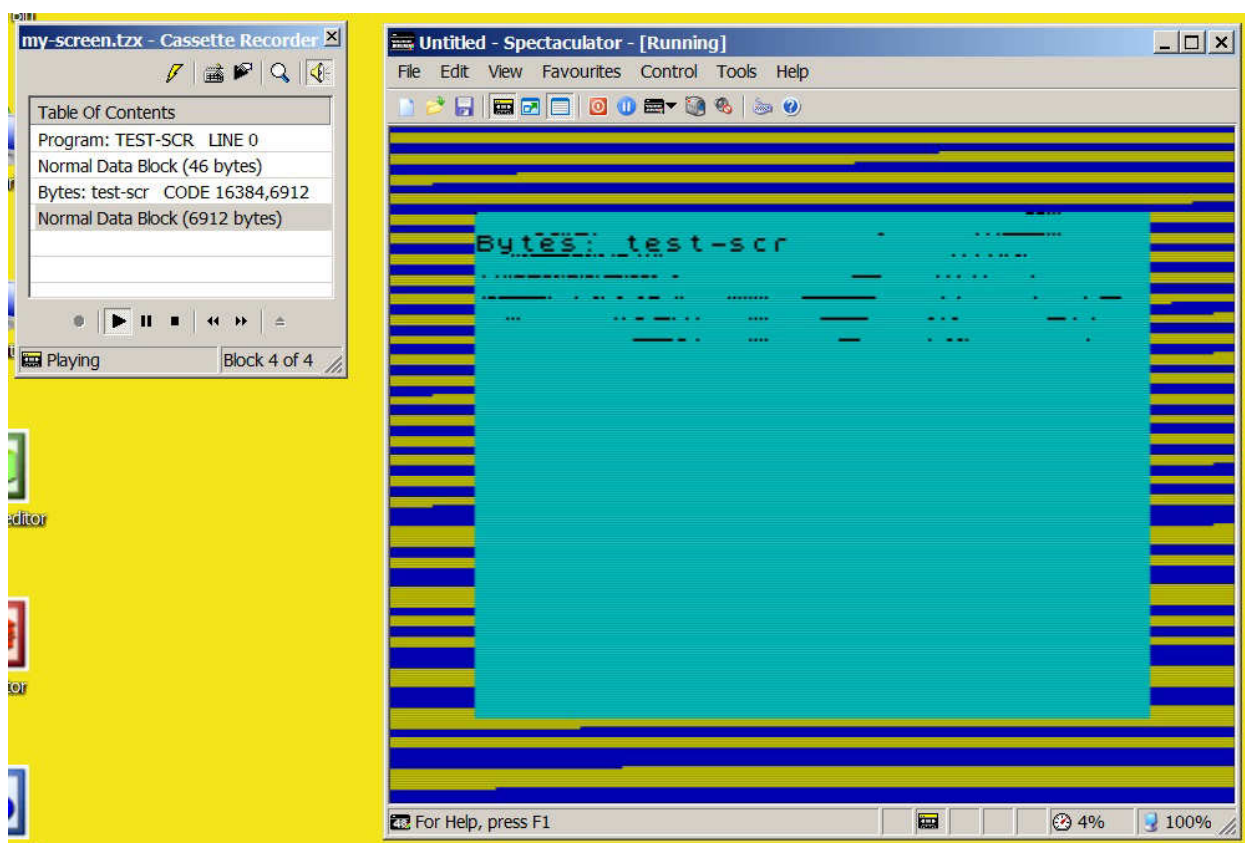


Рис. 4305. Проверка работоспособности склеенного файла программой Tapir на Spectaculator.

После загрузки картинки программа остановится, ожидая нажатия любой клавиши. Полюбовавшись на картинку, нажмите клавишу. Выскочит ОК, затерев нижние строки вашей картинки. Можно посмотреть на свою Basic программу, убедившись, что она на месте.

Теперь рассмотрим, как сделать тоже самое с помощью программы ZX-Blockeditor. Она более сложная и навороченная, и имеет гораздо больше функций, и форматов записи. Рассмотрим только простейшие операции с магнитофонными блоками.

Откройте программу ZX-Blockeditor и перетащите в окно картинку «scr-loader.tzx». В окне редактора появятся не два, а целых 5 блоков. Три из них - служебная информация о типе и версии файла, и контрольная сумма. Эти блоки можно при желании удалить. Они не имеют отношения к байтам данных, загружаемых в ОЗУ Спектрума. Растяните программу так, чтобы виднелись все столбцы таблицы с информацией:

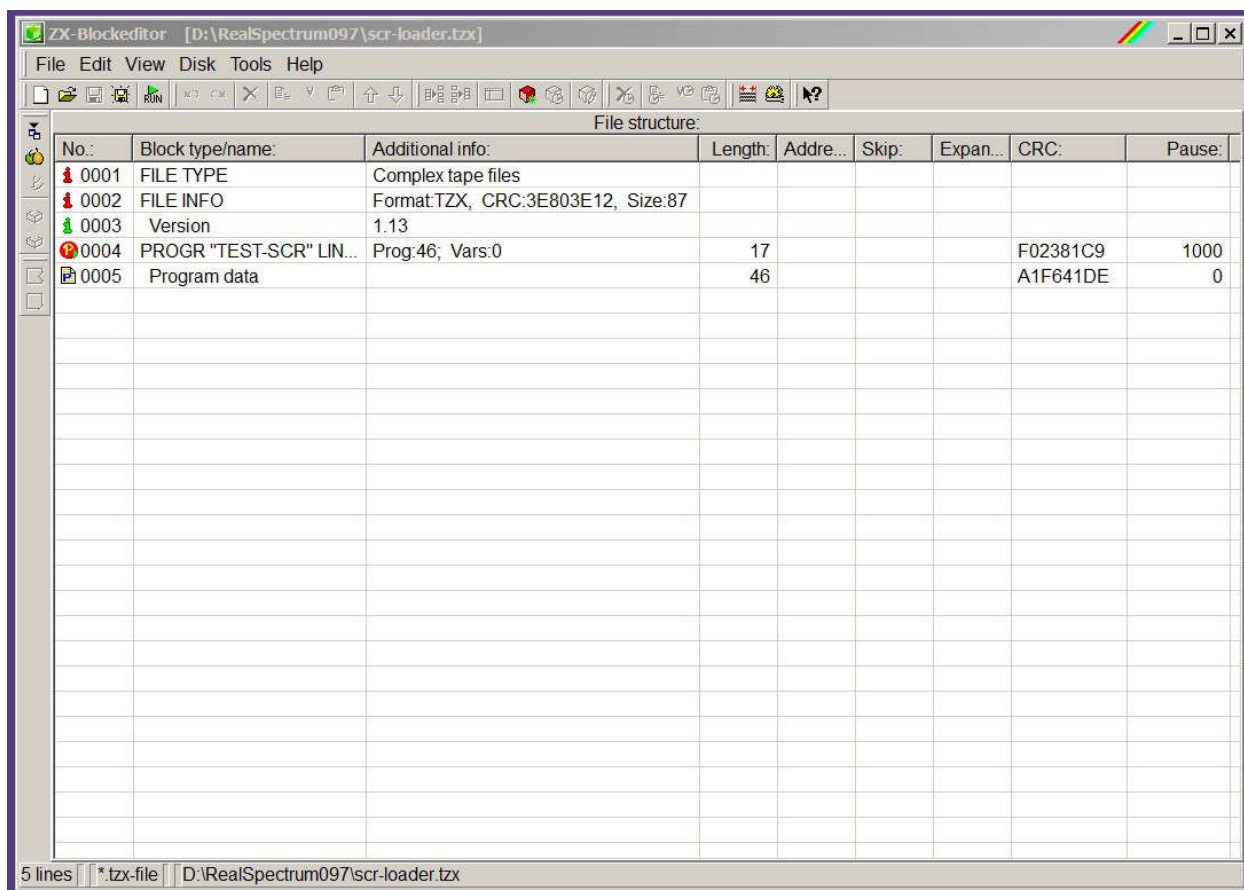


Рис. 4306. ZX-Blockeditor. Просмотр блоков файла «scr-loader.tzx».

Блок 0004 заголовок. В графе «*Additional info:*» видим информацию о длине программы и количестве переменных. Блок 0005 тело программы. Как и в прошлом случае, после блока данных отсутствует пауза, так как он конечный. Потом придется выставить время задержки вручную.

Теперь добавим туда другой блок, картинку «*test-scr.tap*». Сначала выделите блок «0001 FILE TYPE» курсором, чтобы вновь добавленный блок не вклинился в середину текущего. Теперь нажимаем кнопку «*Tools*» на панели программы. Откроются дополнительные опции подменю. Выберите вкладку «*Import file*». Откроется окно выбора файла, для добавления в блок:

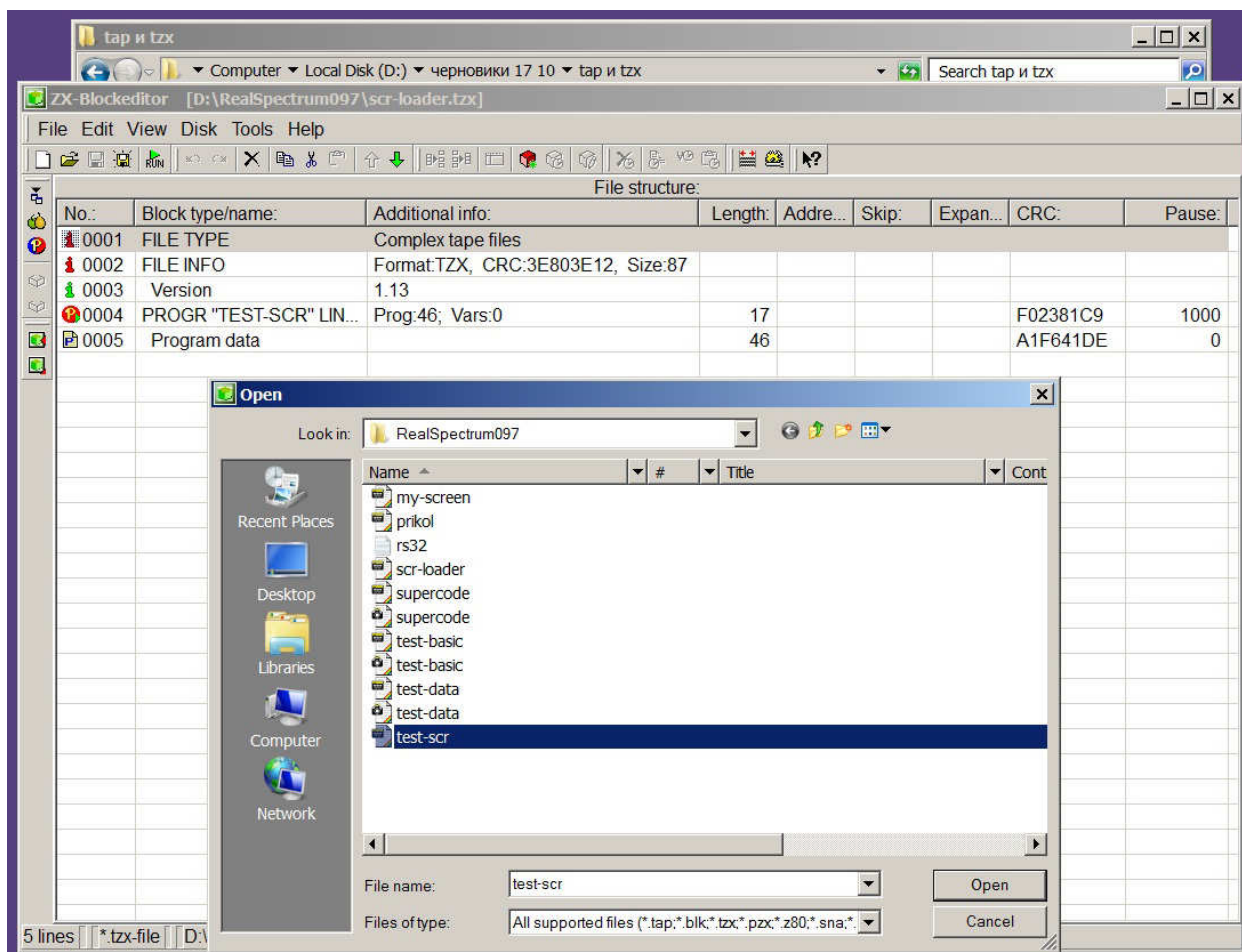


Рис. 4307. ZX-Blockeditor. Открытие файла для добавления блоков к существующему файлу.

Выбрав «test-scr.tap», нажмите «Open». Откроется окно «Import file...» с информацией о блоке «Bytes:». Состоит он из 4 блоков, два из которых также информационные. Растяните окошко по бокам, и вы увидите все графы окна «Import file»:

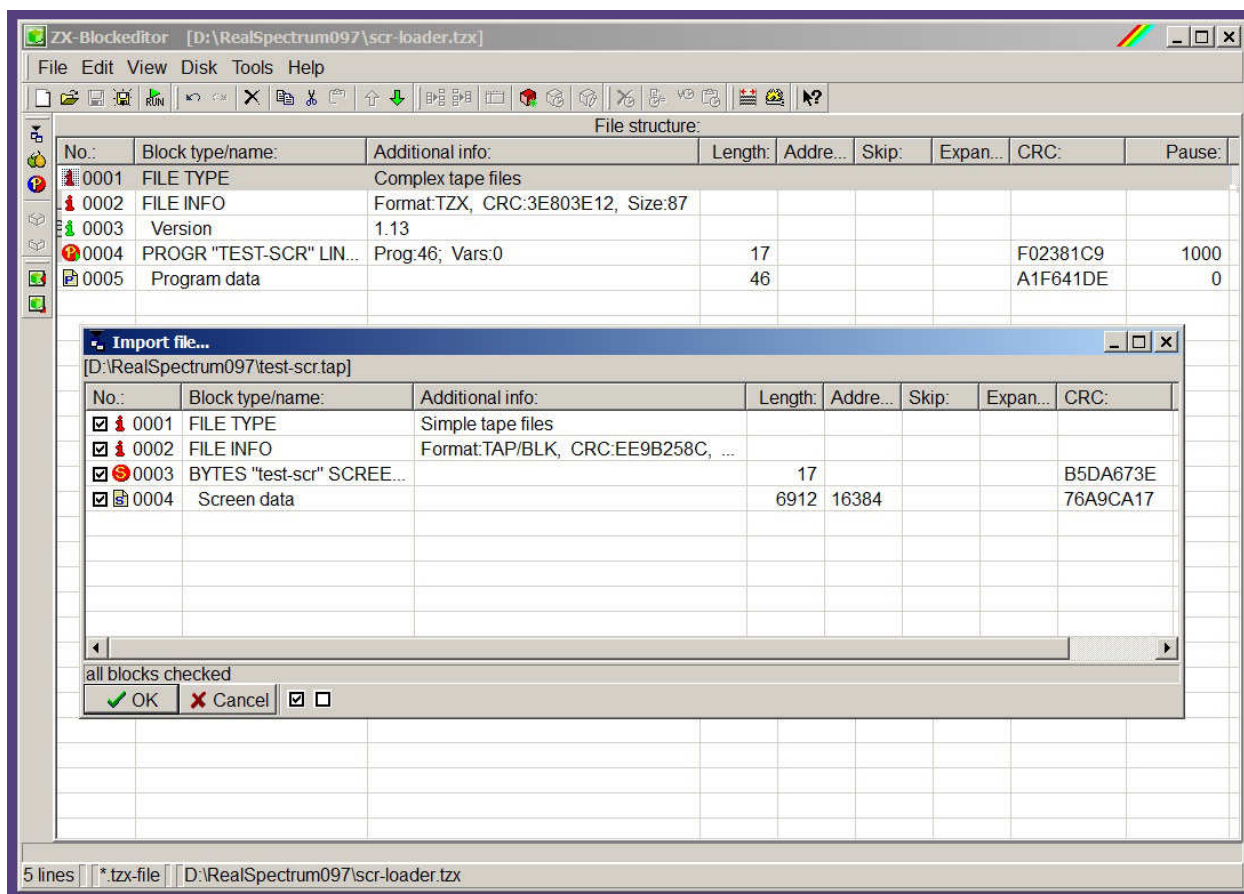


Рис. 4308. ZX-Blockeditor. Окно «Import File». Добавление новых блоков к существующему.

Нажав кнопку «OK» файл «test-scr.tap» добавится в начало списка, потеснив «Program.»:

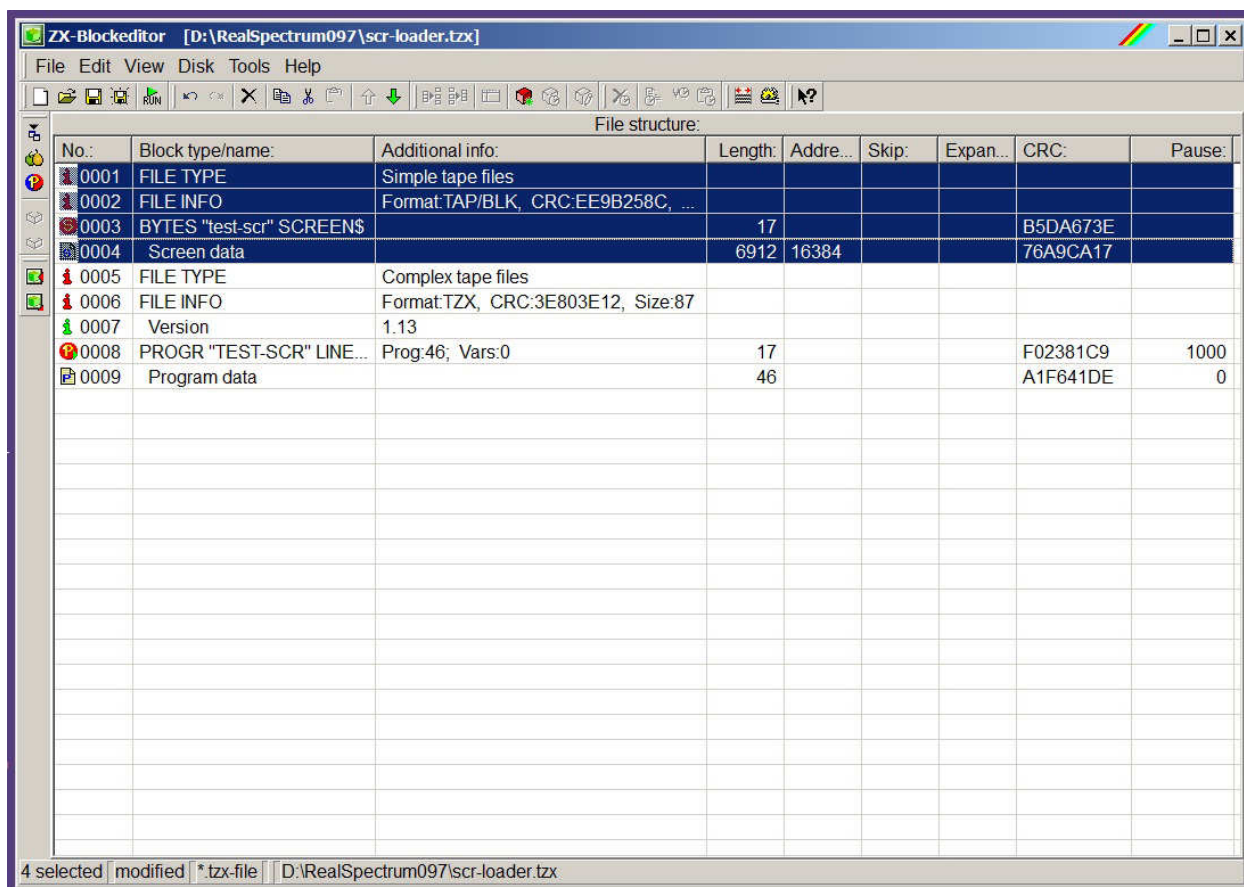


Рис. 4309. ZX-Blockeditor. Склеивание блоков с данными.

Блоки «Bytes:» стоят перед «Program:». Теперь их нужно переместить вниз. Оставьте эти 4 блока выделенными, или если сняли выделение, снова их выделите. Нажмите 5 раз кнопку с зеленой стрелочкой вниз, и группа блоков переместится под «Program data». Теперь выделите все информационные файлы. Удаление выделенных блоков производится с помощью кнопки с крестиком или *CTRL+Y* из окна программы.

Теперь у нас осталось 4 блока. Нужно только выставить паузу между блоками «Program:» и «Bytes:», и записать полученный файл на компьютер. Выделяем блок «0002 Program data» и два раза щелкаем левой кнопкой мыши. Или нажимаем «Edit», а в открывшемся меню выбираем «Modify datablock».

Откроется окошко «Edit datablock», с редактором параметров заголовка. В левой секции в окне параметра «pause in ms:» введите число «1000»:

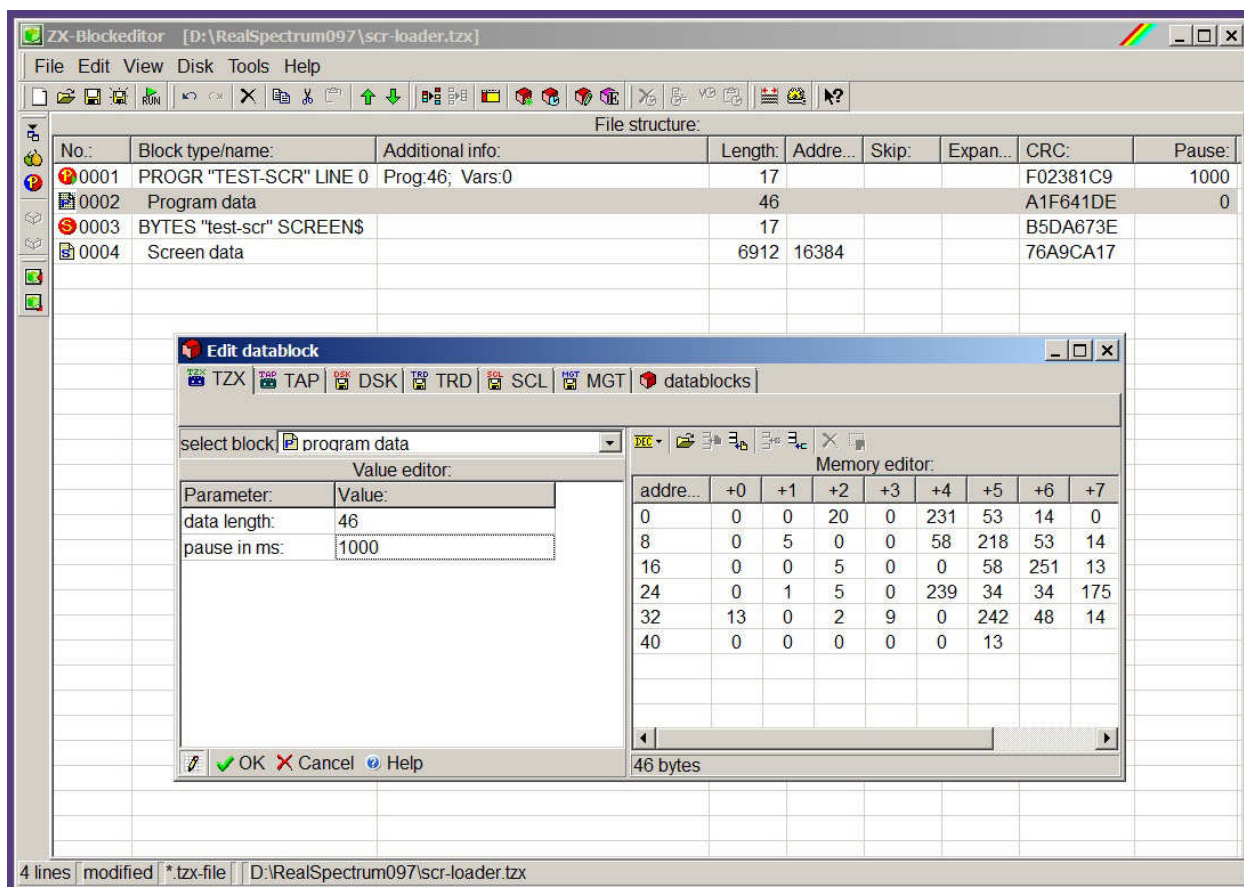


Рис. 4310. Окно «Edit datablock». Выставление паузы между блоком «Program:» и заголовком «Bytes:».

Нажмите «OK» и окно закроется. Теперь нажимаем **CTRL+S** или кнопку «File» и выбираем пункт «Save As...». Откроется окошко записи файла. Сохраните его как «scr-loader2», а в строке «Save as type:» выберите «Complex tape files (*.tZX)». Нажмите «Save» и файл готов. Вы можете проверить работоспособность, не выходя из редактора. На панели программы нажмите кнопочку «Run» с зеленым треугольником. Программа откроет файл в установленном эмуляторе, к которому по умолчанию закреплен этот тип расширения. В моем случае это будет Spectaculator. И программа начнет загружаться обычным образом по **LOAD ""ENTER:**

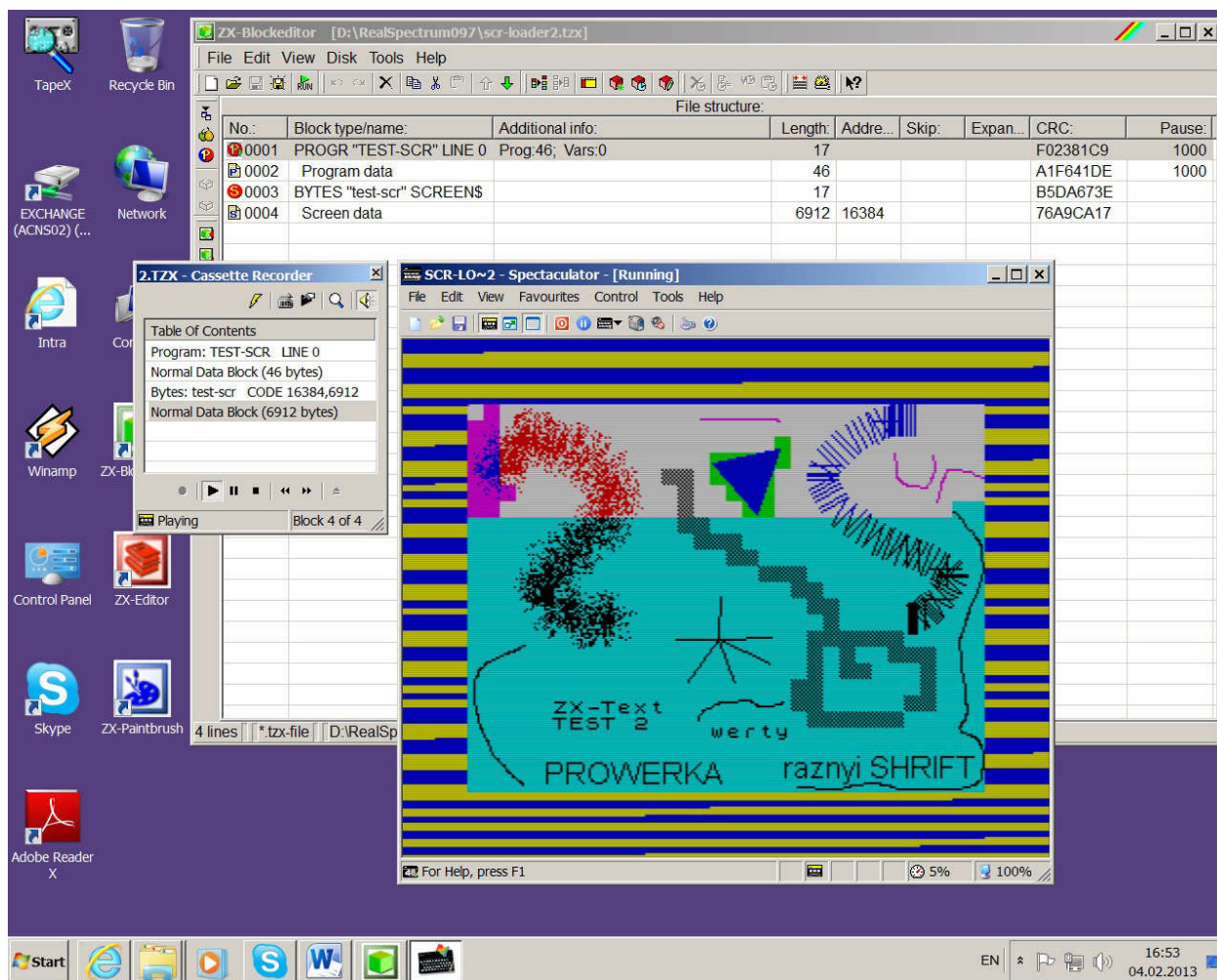


Рис. 4311. Проверка работоспособности программы, склеенной в ZX-Blockeditor.

Проверьте программу на работоспособность, строки BASIC, и закройте эмулятор.

В самом начале 2-й части говорили, что лучше установить все три программы из пакета «ZX-Modules», которые могут взаимодействовать друг с другом. Давайте проверим это на практике. Снова откроем ZX-Blockeditor и созданной программой.

Допустим нам надо отредактировать картинку. Выделим блок «0004 Screen data», нажмем кнопку «Tools». В открывшемся меню выберите «Send block to ...», а в выскочившем подменю выберите «Sent to ZX-Paintbrush»:

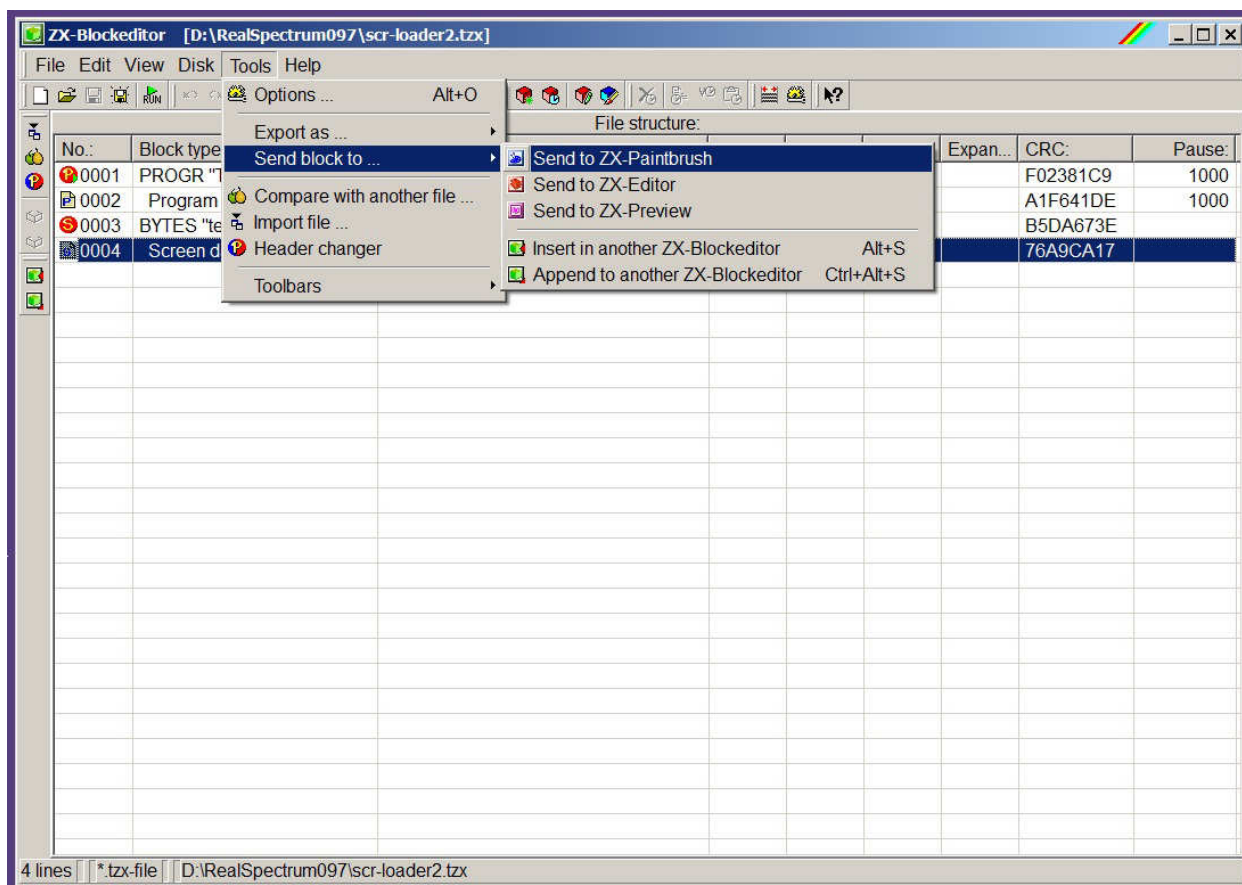


Рис. 4312. Переход из программы в программу в пакете «ZX-Module».

Нажмите левую кнопку мыши и тело блока «Bytes:» откроется в виде картинки в «ZX-Paintbrush». Эта опция есть у всех программ этого пакета, связывая их между собой.

Глава 4. Создание звуковых .wav файлов с данными.

Краткое содержание: проигрывание магнитофонных файлов в Tapir, преобразование магнитофонных файлов в звуковые.

Если вас есть настоящий Спектр, то загружать программы с магнитофона не самый оптимальный вариант. Оригинальные кассеты уже давно стали музейными экспанатами, а запись на них скорее всего давно повредилась или размагнитилась. Поэтому наилучшим вариантом загрузки программ будет с помощью компьютера или плеера. Можно придумать несколько разных способов. Например, проигрывать tap/tzx файл в эмуляторе, параллельно подсоединив к выходу звуковой карты IBM-совместимого настоящий Спектр. Но можно сделать более экзотичнее. Создать собственный .wav файл, а затем воспроизводить его в любом проигрывателе, например Winamp, отредактировать или сделать *.mp3.

Для примера создадим *.wav файл. Наиболее просто это можно сделать в программе Tapir. Откроем программу и перетащим наш созданный файл «my-screen.tzx» в левое окно редактора. В меню «Left». откройте меню «Play to WAV». Откроется окно «Sample rate», где вам предложат выбрать частоту звука. Нажмите 44100 Hz:

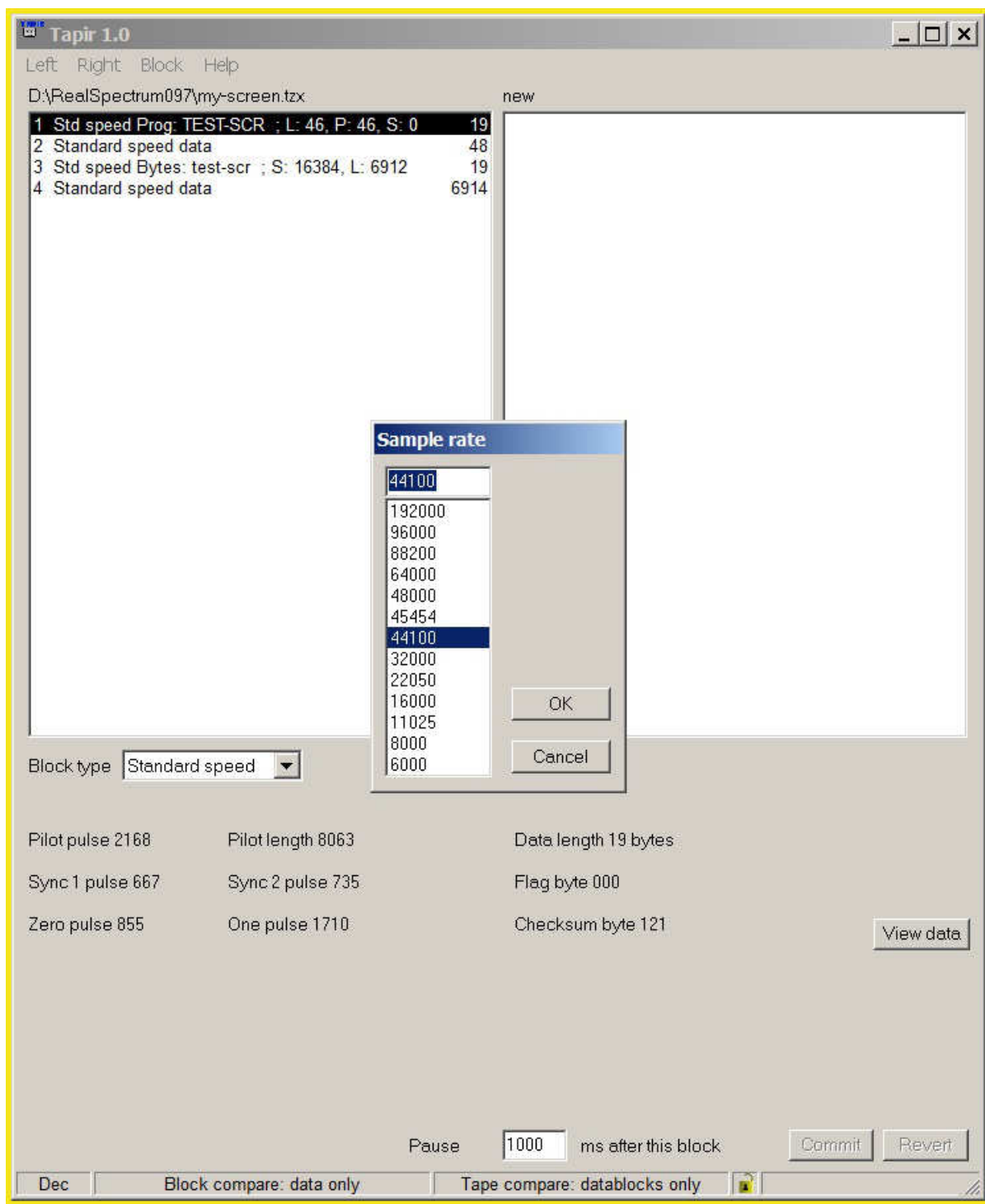


Рис. 4400. Окно «Sample rate». Выбор частоты будущему .wav файлу.

Нажмите «OK». Откроется окно «Save As», в котором нужно набрать имя файла, также, не забыв расширение «my-screen.wav». Произойдет конвертация и у нас, получится .wav файл с картинкой, который можно проиграть, в winamp:

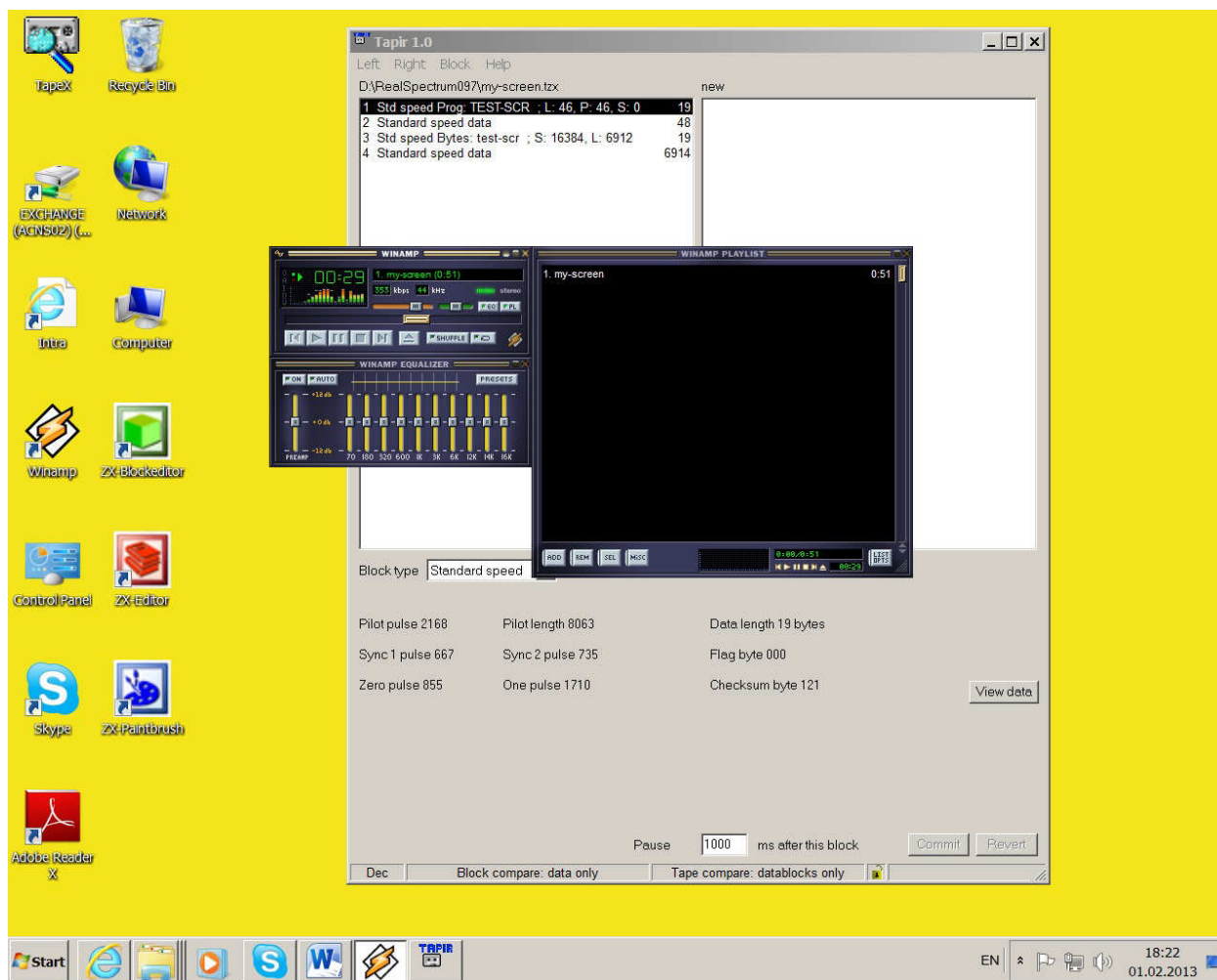


Рис. 4401. Прои́грывание файла «my-screen.wav» в winamp.

Закрываем программу. Файл готов. Для загрузки в ZX-Spectrum, можно воспроизводить любыми способами: с компьютера через звуковую карту, флешки, с портативного CD, DVD плеера, или даже с магнитофона, предварительно записав файл на кассету.

Глава 5. Создание блока кодов из фрагмента данных.

Краткое содержание: преобразование данных в сигнал, редактирование данных в Тариг, корректировка контрольной суммы блока, добавление пилот-тона к данным.

Допустим, имеется фрагмент дизассемблированной программы. Из этих данных требуется создать *.tap / *.tzh файл. Принцип создания файлов данных уже рассматривали на экспериментах с ПЗУ. Теперь попробуем создать программу в машинных кодах, и с помощью Тариг создать простой блок загрузки.

Предположим, имеется простейшая программа, которая при запуске по `RANDOMIZE USR` выводит зеленую рамку:

```
LD A, 4
OUT (254), A
RET
```

В десятичном варианте она будет выглядеть так: 62, 4, 211, 254, 201

Откроем Hexedit и создадим пустой файл, длиной 7 байт, как описывалось в предыдущих главах. Первым байтом введем маркер «255», который будет указывать, что это блок кодов. После него введем последовательно эти значения, а самый последний байт также оставим пустым для контрольной суммы. Пусть вычислит ее программа:

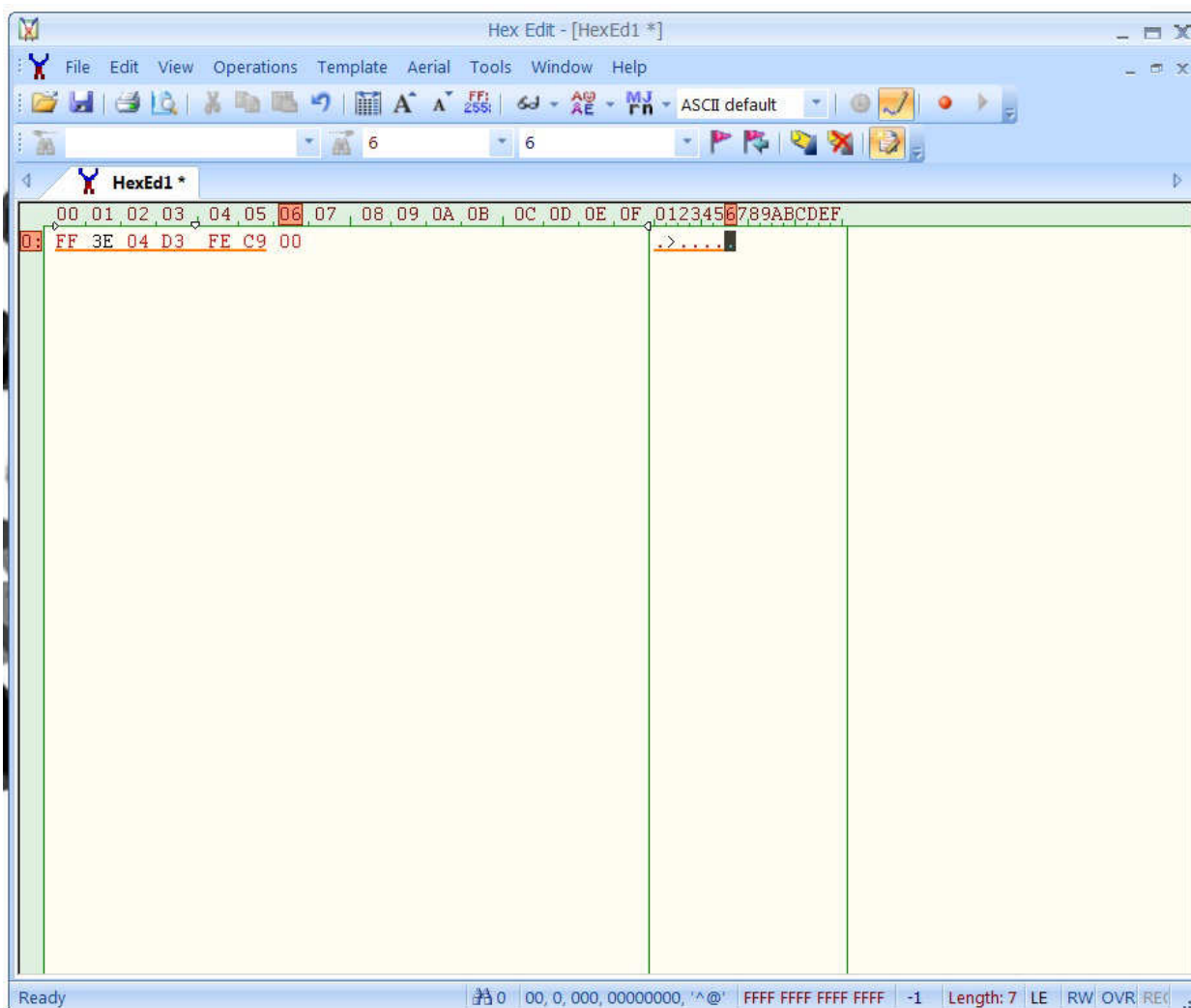


Рис. 4500. Редактор Hex Edit. Создание блока данных.

Сохраним файл, под именем «no-title» (без расширения). Теперь эти данные требуется преобразовать в блок данных. До этого мы занимались только редактированием готовых блоков, а теперь придется создать собственный блок, и ознакомиться с настройками сигнала и структурой файла. Откроем Tapir, нажмем кнопку «Left». Из открывшегося меню выбираем «Add block», а из выскочившего большого подменю «Data from file»:

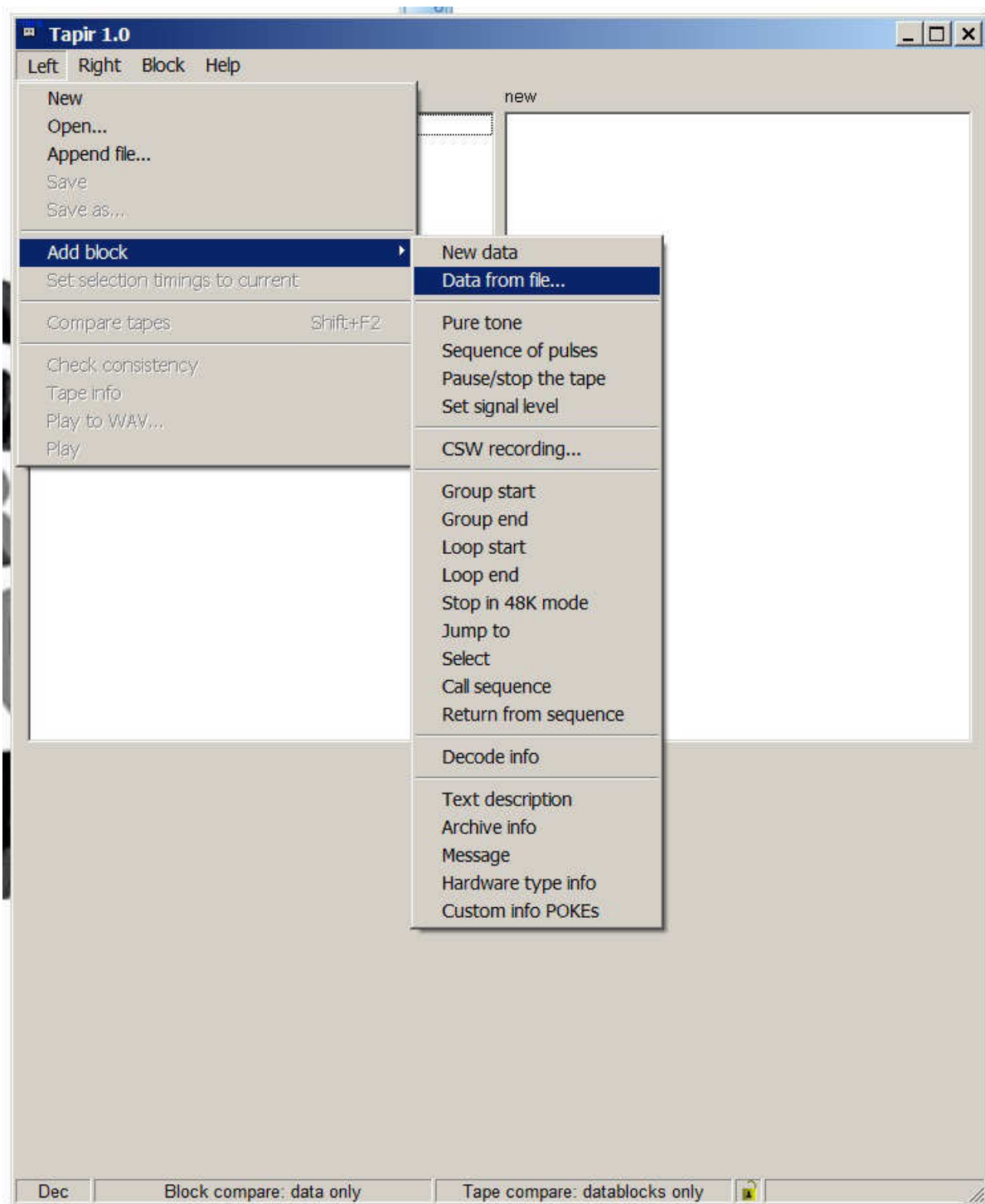


Рис. 4501. Tapir. Создание искусственного блока из данных файла.

Нажав кнопку мыши на выделенном пункте, выскочит стандартное меню «Open», где предложат открыть любой бинарный файл. Выберем созданный в Hexedit файл «no-title» и нажмем «OK». В левом окне появится блок «Pure data» и справа его размер 7 байт. Выделите его мышкой, он покраснеет, и внизу отобразятся его характеристики:

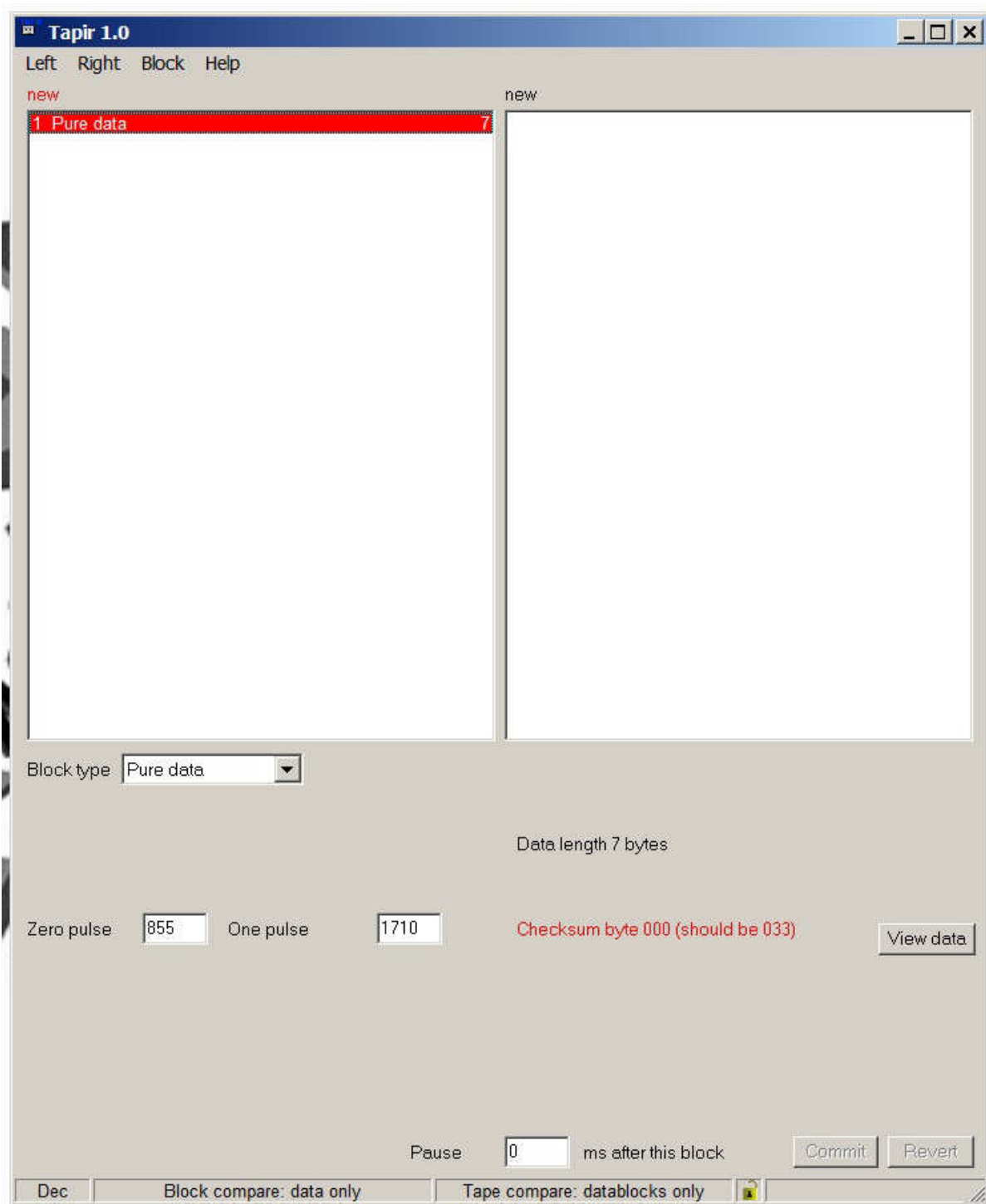


Рис. 4502. Tapir. Процесс создания блока данных. Неверная контрольная сумма.

На этот раз в окошечке «*Block Type*», вместо «*Standart speed data*» мы видим «*Pure data*», а для редактирования доступны два параметра: сигнал, который соответствует биту «0» (*Zero pulse*), и биту «1» (*One pulse*).

И действительно, включив предварительную прослушку во встроенном плеере (*Left* → *Play*) услышим только сами данные без пилота. Обратим также внимание на красное сообщение об ошибке **Checksum byte 000 (should be 033)**. Байт контрольной суммы блока «0», а должен быть «33». Поэтому следующим шагом, будет исправление контрольной суммы. Нажмите кнопочку «*View data*» и откроется окно редактора. В нижнем левом углу вы увидите окошко с надписью «*HEX*». Наведите на него и нажмите левую кнопку мыши. В окошке загорится «*DEC*» и все числа в окне преобразуются в десятичный формат.

Обратим внимание, в редакторе показаны всего 5 байт, хотя мы делали 7. Куда пропали два крайних? Ко всему прочему значения не отредактировать.

На самом деле два байта программы скрыты, и нам нужно их открыть. В правом верхнем углу снимите галочки с опции «*Hide flag byte*». Перед программой сразу появится дополнительный байт «255», то есть маркер блока данных. Далее снимите «*Hide checksum byte*». Теперь откроется последний байт «000», стоящий за нашей программой. Он то и есть зарезервированный байт для контрольной суммы. Вместе со снятиями галочек, появилась и возможность редактирования значений данных. Теперь давайте исправим контрольную сумму.

Наведите курсор на средний нолик и нажмите кнопку. Байт будет выделен двухцветным курсором «000» (по краям синий, по центру красный). Красная часть курсора будет означать, что именно с этой части байта можно начинать редактировать все значение. Теперь отредактируем его введя значение «33». Красный курсор переместится на первую цифру. У вас должно получиться так:

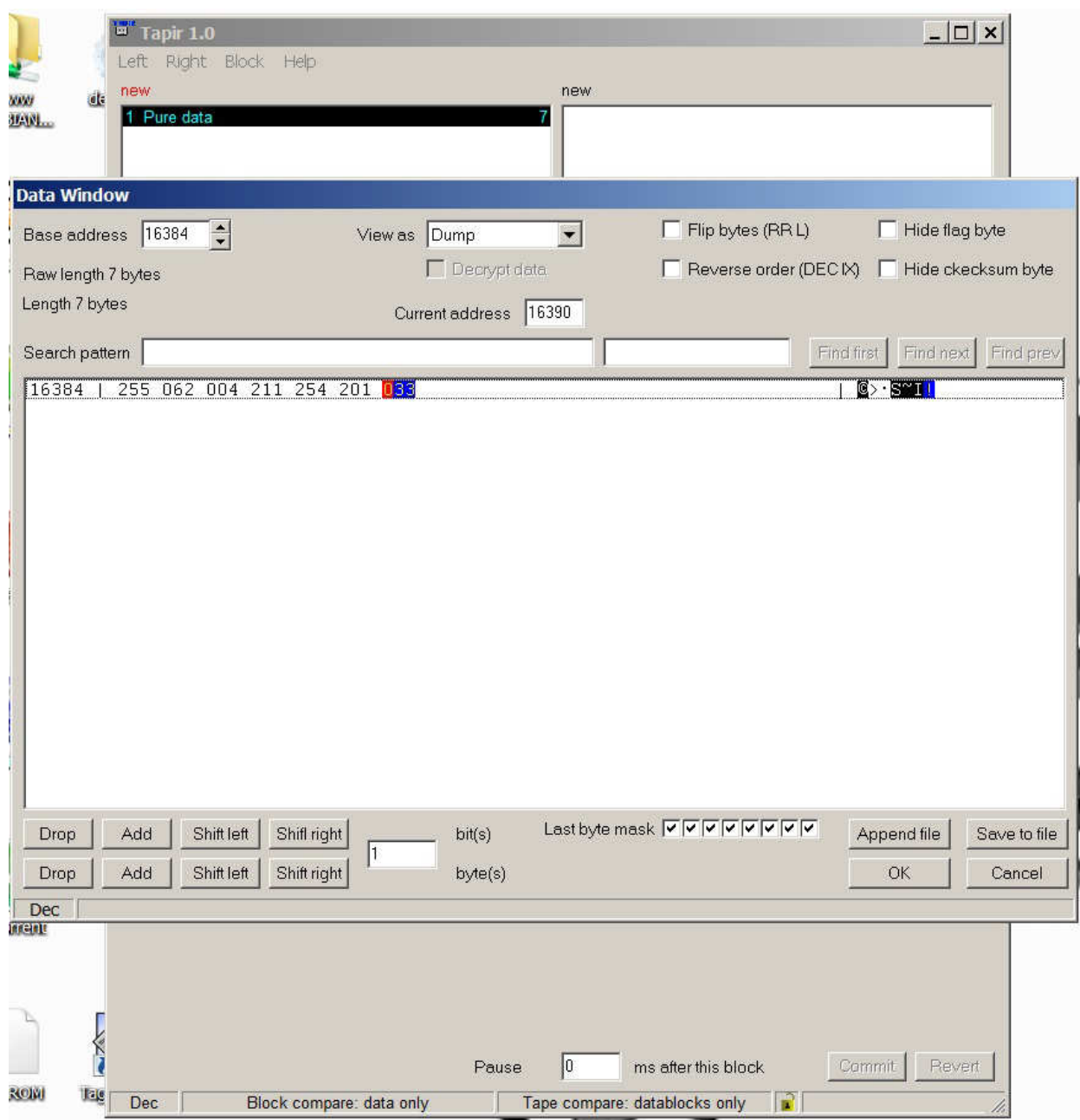


Рис. 4503. Окно Data Window. Корректировка байта контрольной суммы блока данных.

Нажмем «OK», окно закроется, и сообщение об ошибке исчезнет. Теперь надпись «checksum byte 033», а также выделение самого блока, станет черного цвета. Все в порядке. Поэтому следующим шагом нужно преобразовать блок, создав и подсоединив к нашим, данным пилот-тон с синхроимпульсом.

Для этого в выпадающем меню «Block type» выберите «Standart speed». В нижней части появятся знакомые параметры сигнала стандартной загрузки. Теперь нажмите кнопку «Commit», и блок преобразуется в нормальный формат данных спектрума. Теперь он вместо «Pure Data» стал «Standart Speed Data». Но в окошке «Pilot length», по прежнему, стоит значение «8063», то есть длинный фрагмент, предназначенный для заголовка. Снова выберем пункт «Standart Speed Data» и значение длины пилот-тона скорректировалось до стандартного «3223». Нажмем еще раз «Commit» и блок будет полностью сформирован:

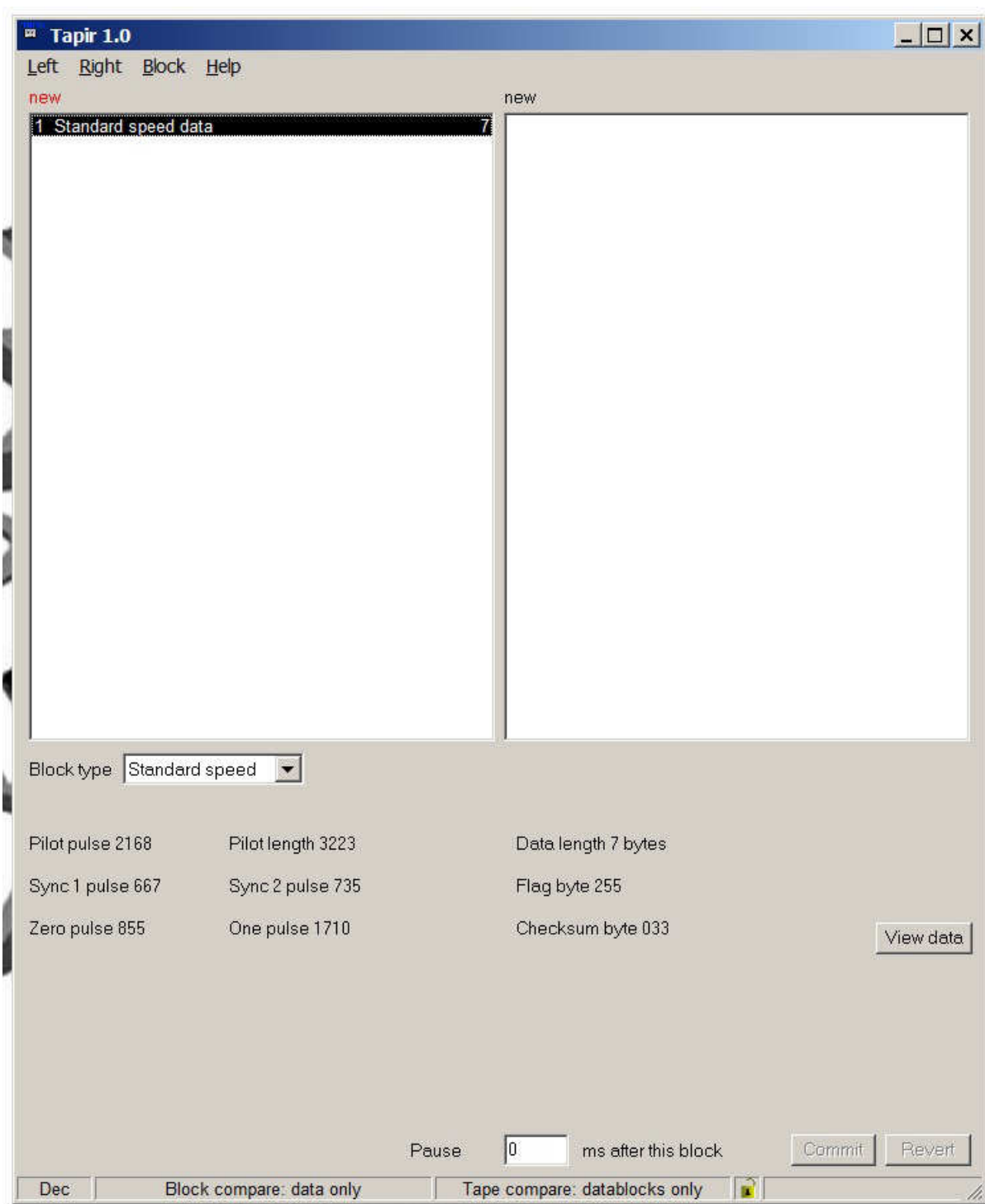


Рис. 4504. Tapir. Полностью сформированный блок данных без заголовка.

Теперь сохраним созданный блок под именем «no-title.tzx» (Left → Save as... не забыв добавить расширение .tzx). Нажмем «Save». Таким образом, мы записали блок данных с машинным кодом, без привязки к адресу ОЗУ. В окне «Data Window» есть встроенный дизассемблер. Например, мы хотим посмотреть, как выглядит получившаяся программа на ассемблере. Снова нажмите «View Data». В окошке «View as» выберите «Disassembly». Программа отобразится на Ассемблере:

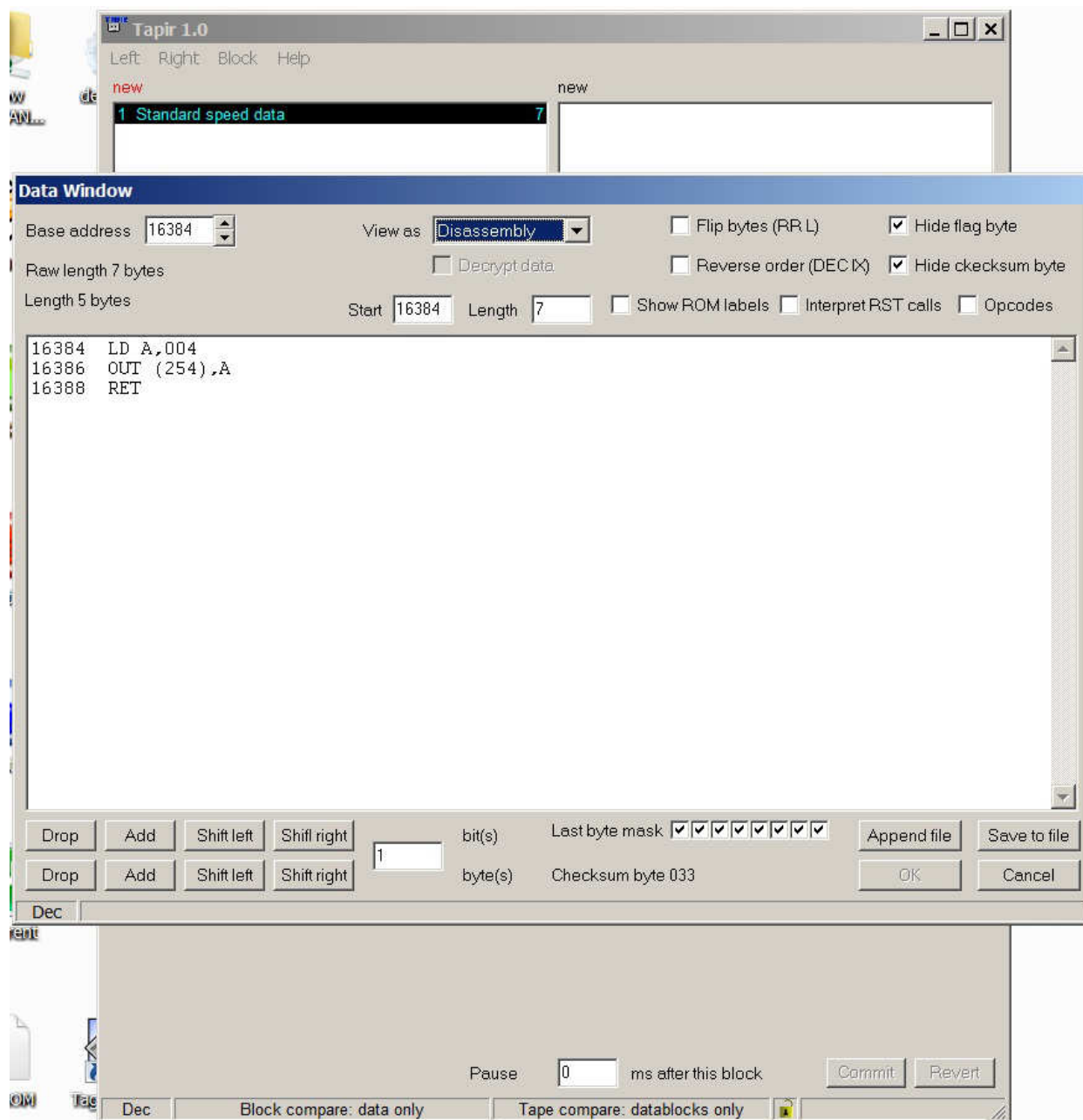


Рис. 4505. Окно Data Window. Просмотр данных в режиме «Disassembly».

Установив в режиме «Dump» курсор на нужный адрес, можно дизассемблировать выборочный кусок программы. Установка галочки «Opcodes» позволяет включить / выключить в дизассемблированную программу десятичные / шестнадцатиричные коды команд. После просмотра закройте все окна программы. Теперь осталось проверить работоспособность блока. Но перед этим нужно создать ему искусственный заголовок. Об этом в следующей главе.

Глава 6.

Создание искусственного блока заголовка.

Краткое содержание: создание блока данных в Tapir, добавление данных в блок, редактирование параметров заголовка.

Блок данных без заголовка из файла с данными создавать научились. Теперь попробуем создать искусственный заголовок под свою программу. В Tapir есть возможность создать блок данных или заголовок с нуля. Попробуем это сделать.

Откройте программу нажмите «Left», в меню выберите «Add block», а подменю самый первый пункт «New data»:

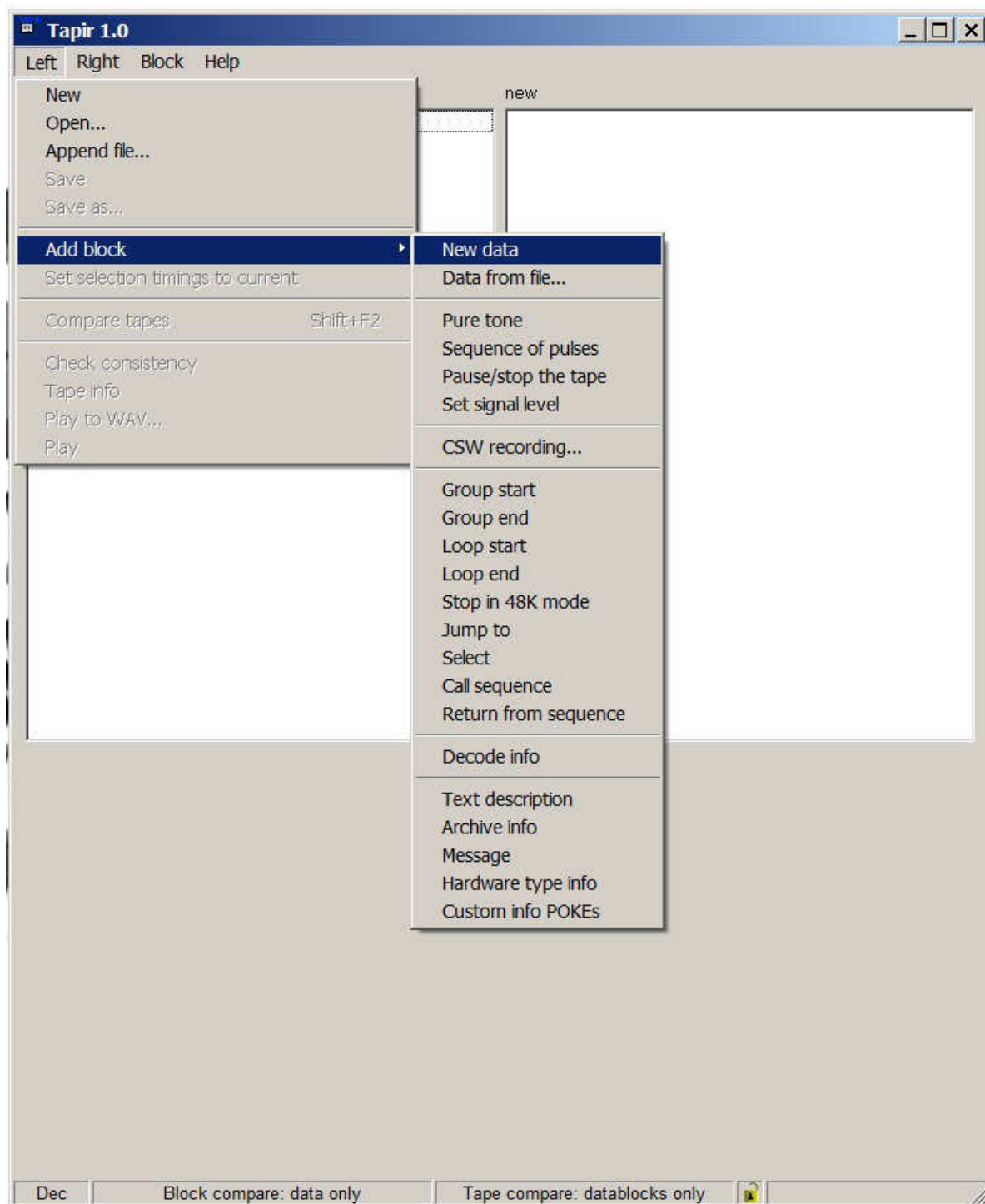


Рис. 4600. Tapir: Создание нового блока данных под заголовок.

После нажатия кнопки создастся пустой блок «Pure data», длиной 0 байт. Выделите его и нажмите «View data». В открывшемся окне «Data window» он будет пустым. В выпадающем меню «View as», доступны всего 5 функций просмотра. Меню «Header» нет. Переключите просмотр данных с «Hex» на «Dec». Теперь нужно создать 19 байт данных.

Для этого в режиме Dump нажмите самую нижнюю кнопочку Add. В левом верхнем углу появится байт 000 с красным курсором. Таким образом, можно добавлять байты, создавая файл. Внизу, справа от рядов кнопок есть белое окошко, в котором стоит число 1 с подписью «bit(s)» и «byte(s)». Чтобы не тыкать кнопку «Add» еще 18 раз подряд, введите в это окошко число «18» и нажмите «Add». Появятся сразу 18 дополнительных байт, которые расположатся следом за первым. Таким образом, мы создали пустой файл, длиной 19 байт. Теперь нажмите на черный треугольничек меню «View as» и сразу заметите, что появился самый нижний пункт меню «Header». При создании файла 19 байт, активизировался новый пункт меню:

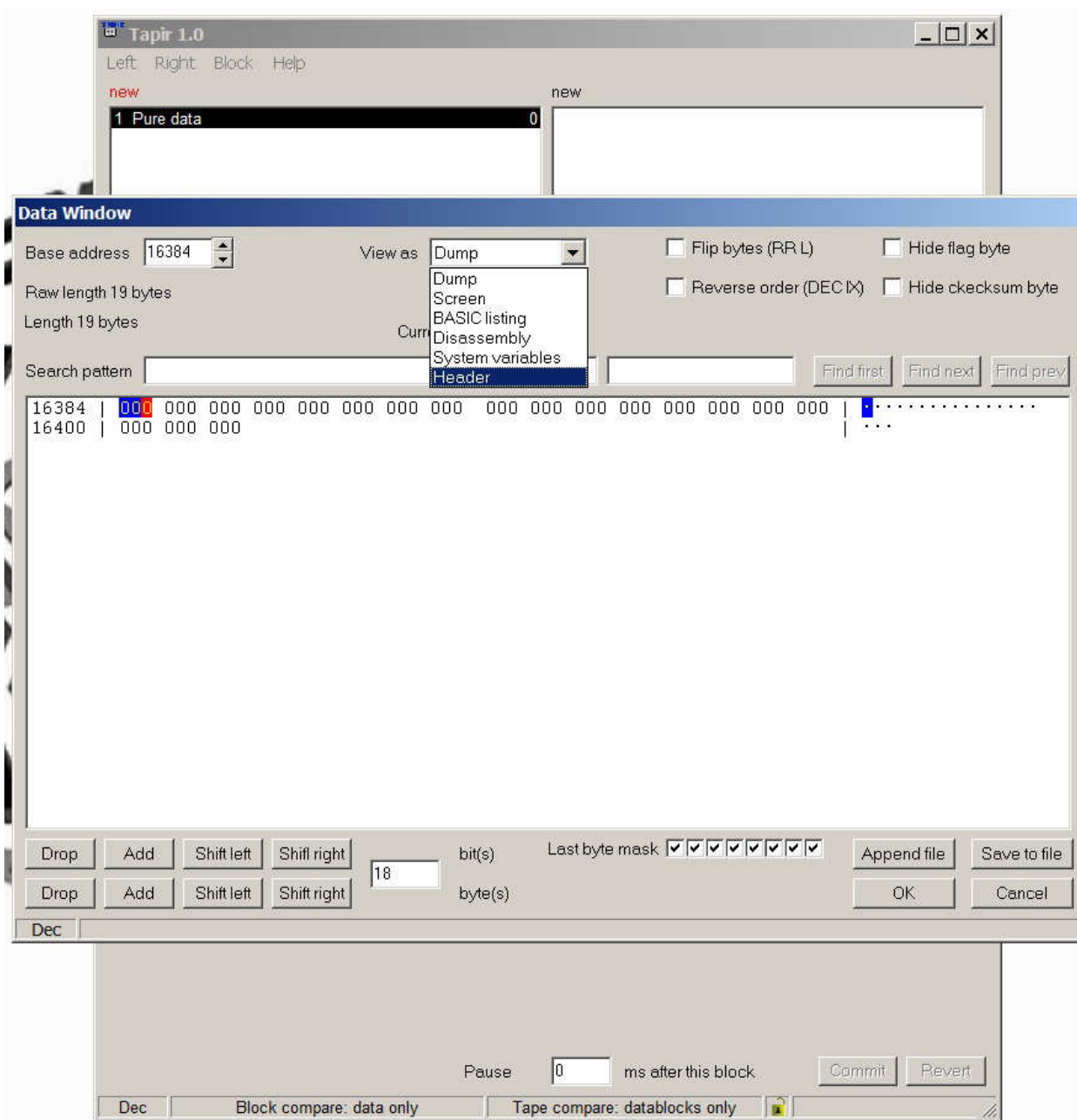


Рис. 4601. Окно Data window. Создание 19 байт данных для заголовка.

Нажмите на него, и окно превратится в редактор параметров заголовка. В графе «Type» из выпадающего меню выберите «Bytes:». Укажите спектрмское имя этому блоку, например «bl-loader». В окне «Start», указываем стартовый адрес будущего блока кодов (не заголовка), который будет к нему подсоединен. Наберите «30000». А рядом параметр «length» длина будущего блока. Следует указывать чистую длину блока, без двух служебных байт (маркера и контрольной суммы):

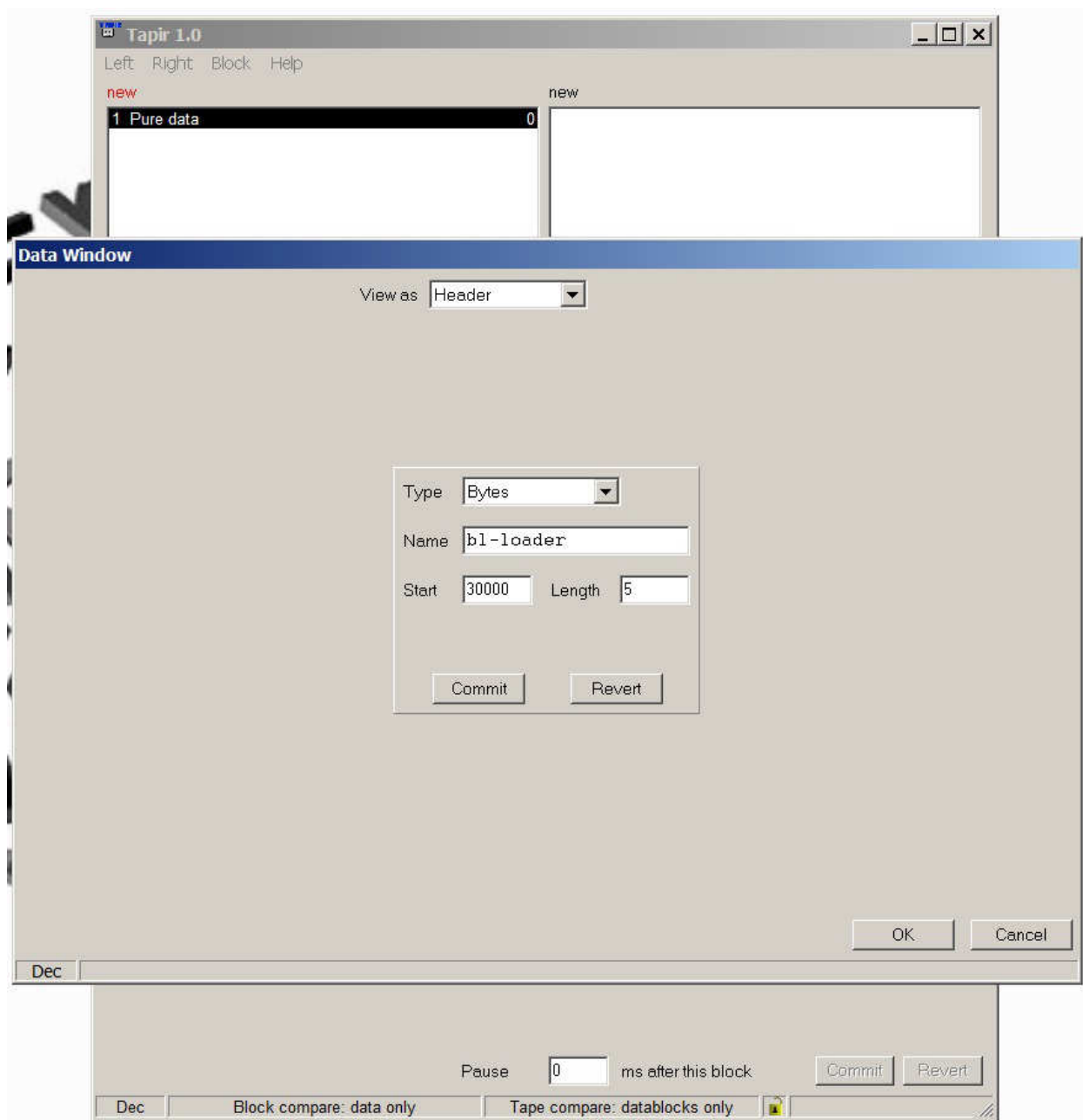


Рис. 4602. Окно «Data window». Формирование заголовка из данных в режиме Header.

Нажмите «Commit» в маленьком окошке. Кнопочка посереет. Затем нажмите «OK» внизу большого окна, и мы выйдем в основную программу.

Но блок пока еще у нас «Pure data». Подсоединим к нему пилот-тон. Для этого в меню «Block type» выберите «Standart Speed», а затем «Commit». В левом окошке у нас появится полноценный блок заголовка «Bytes:» со всеми заданными параметрами и автоматически созданной контрольной суммой в последнем байте:

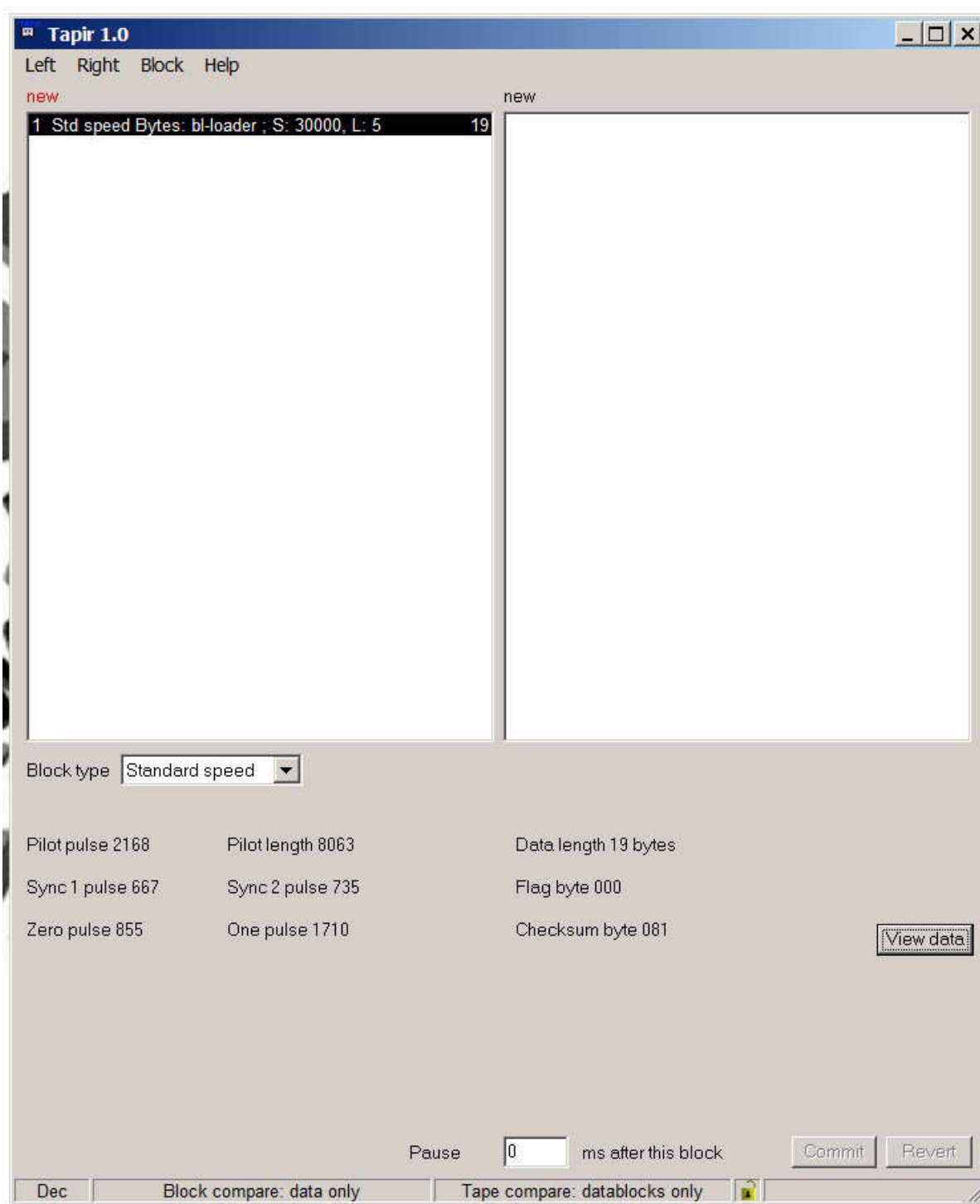


Рис. 4603. Tapir. Полностью созданный заголовок для блока «Bytes:».

Шаблон заголовка готов, осталось только его записать. Запишем его под именем «bl-loader.tzx» и выйдем из программы.

ЧАСТЬ 5.

Примеры создания сложных программ с помощью эмуляторов и утилит.

В этой части книги будем ставить эксперименты, создавая разные сложные варианты процесса загрузки файлов, а также управления BASIC системой из машинных программ. Некоторые программы были придуманы мной. В чем-то натолкнула на идеи

старая литература, где-то фрагменты программ были взяты из ранних номеров журнала ZX-Ревю, переработанные с учетом современных реалий работы с эмуляторами и утилитами. Цель главы, не столько создание оригинальных программ, сколько методика гибридной работы с эмуляторами и утилитами одновременно. Поиск оптимальных путей решения проблем, и постепенная доработка программ разными средствами, облегчающих достижения заветной цели. Так как Спектрум, это давно изученный простой компьютер, то в своих программах я мог неумышленно повториться, не подозревая, что этот метод где-то уже описывался.

Глава 1.

Создание длинного правильного блока «Program:» с невидимыми данными.

Краткое содержание: расчет длины блока «Program:», подгонка адресов запуска, монтирование цепочки шестнадцатиричных данных, склеивание данных блоков «Program:» и «Bytes:», подмена строк одинаковой длины, загрузка невидимых данных.

Во 2-й части книги рассматривали автозапускной блок «**Bytes :**» с полностью работоспособной программой, и корректным выходом в интерпретатор BASIC. А теперь попробуем представить, что нужно создать программу с противоположными свойствами. Стартует блок «**Program :**», но по звуку слышно, что идет загрузка картинки. После загрузки блока, картинка появляется на экране. Вы открываете, думая там увидеть черный экран, нулевую строку огромной длины, с «мусором», рычание и зависание.

Открыв ее, вы ничего подобного не найдете. Белый экран и единственная строчка с командой **1 REM**. Удаляете строчку, и ничего не происходит. Пишете свои строки программы, и снова все в порядке. Компьютер нормально работает, и нет никакой защиты. Все открыто и прозрачно.

Давайте попробуем создать такой блок. Для начала составим программу, которая будет выводить эту самую картинку на экран, а заодно посмотрим, сколько она займет места в памяти. Известно, что длина блок «**Program :**» содержит чистые данные программы, если эта программа не запускалась, или с последними переменными, если она хоть один раз выполнялась. Набрав нужную программу, данные можно вычислить теоретически по формуле:

$$\text{Расчетная Длина} = \text{VARS} - \text{PROG}$$

где,

VARS = значению ячеек (23627) + (23628)*256

PROG = значению ячеек (23635) + (23636)*256

Проверим это на практике. Наберем в Spectaculator подготовительный загрузчик для запуска картинки, который будет существовать во время загрузки, а после старта машинных кодов, исчезнет. Он будет состоять из единственной строки:

```
1 RANDOMIZE USR 1
```

Это все что требуется для программы. Откроем «*Debugger*» и посмотрим, сколько байт заняла строчка:

00, 01, 10, 00, 249, 192, 49, 14, 00, 00, 01, 00, 00, 13

Посмотрев системную переменную **VAR\$**, видим, что ее адрес указывает на 23769, в которой стоит маркер «128». Переменная **E-LINE** указывает на ячейку «23770», в которой маркер «13», следом за которым стоит еще одно значение «128», означающее конец этой области. Другие переменные указывают на адрес «23772», где видны остатки мусора от предыдущего ввода строки.

После строки BASIC обязательно должен стоять маркер, иначе все что стоит дальше, будет восприниматься как следующая строка, поэтому в конце строки поставим «128». (Для надежности в редактируемую строку 13, 128)

Давайте проверим, действительно ли загружаемая программа будет длиной 14 байт. Создадим в Realspectrum «scrprog.tzx» и загрузим в Spectaculator'е для проверки:

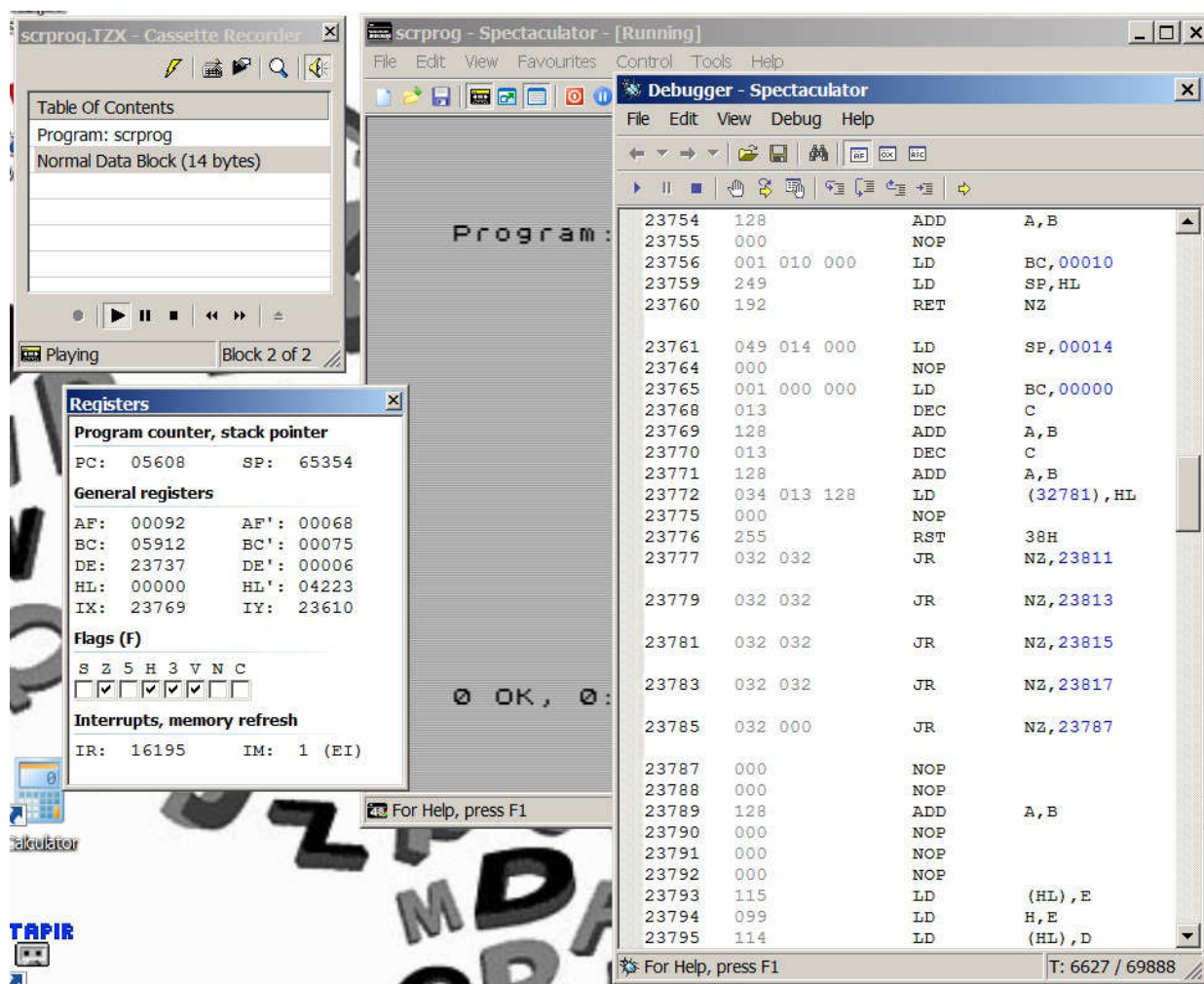


Рис. 5100. Spectaculator. Проверка расчетной длины программы и системных переменных на практике.

Так оно и есть. Программа заняла ровно 14 байт, а после загрузки следом за ней выстроились маркеры перехода «128», пустующих областей BASIC системы.

Но если вы посмотрите программы посложнее, где есть переменные, циклы и прочее, то вы увидите, что длина блока программы включает в себя еще и обширную область переменных **VAR\$**, вплоть до области вводимой строки **E-LINE**. (О том, как перемещать сложные программы разной длины из произвольных областей памяти, накладывая друг на друга без последствий, будет описано в следующей главе.)

Зная примерную технологию работы программы, можно создать искусственный заголовок, который будет указывать на блок нужной длины, или модифицировать созданный. Затем к нему подсоединить блок данных с программой расчетной длины, с искусственной строкой BASIC, отделенной по всем правилам маркерами переходов. Они нужны чтобы не подцепить из области памяти данные, которые BASIC распознает, как

строки и попытается вывести на экран фактически все ОЗУ в виде листинга знаков вопросов, сопровождаемый рычанием, с последующим зависанием.

Наша программа будет гибридной, состоять из BASIC строк вперемешку с машинными кодами, и выглядеть следующим образом:

ORG 23755

DEFB 00, 01, 10, 00, 249, 192, 49, 14, 00, 00, 220, 92, 00, 13
DEFB 128, 13, 128 ; 1 RANDOMIZE USR 1 [23772]

LD HL, stroka
LD DE, 23755
LD BC, 17
LDIR

LD HL, kartinka
LD DE 16384
LD BC, 6912
LDIR
RET

stroka: DEFB 0, 1, 10, 0, 234, 80, 117, 115, 116, 111, 116, 97, 33, 13
DEFB 128, 13, 128 ; 1 REM Pustota!

kartinka: DEFB 255

Последний байт делаем к качестве опознавательного знака начала картинки. В качестве картинки можно использовать «*scr-test.tap*», к которой спереди подсоединим всю эту программу, немного потеснив картинку.

Чтобы быстро перевести и рассчитать адреса подменяемой строки и картинки, можно воспользоваться эмулятором EmulzWin. Для этого откройте эмулятор, окно ассемблера, и скопируйте туда программу с метками (*CTRL+C*), написанную выше. Скомпилируйте программу, и откройте «*Debugger*», прокрутив строки до адреса 23755:

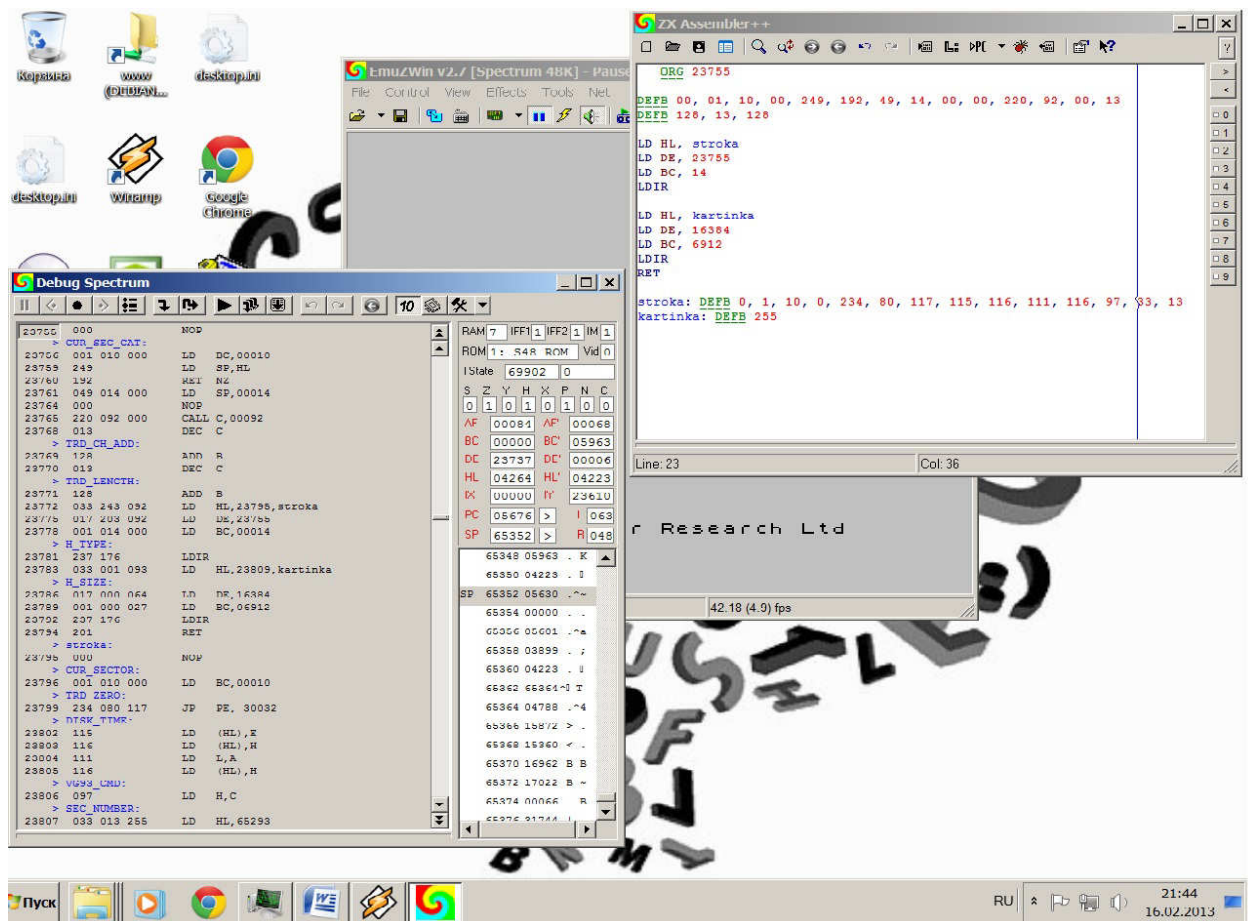


Рис. 5101. EmulZWin: В отладчике видна скомпилированная программа.

Посмотрите внимательно на программу. Она скомпилировалась и заняла адреса с 23755 по 23709 включительно. Следовательно, ее длина будет 55 байт. Машинная программа начинается с адреса 23772. Переменная «stroka» имеет адрес 23795, а переменная «kartinka» - 23809. Теперь можно для себя пометить точные адреса размещения всех данных.

Отжимаем кнопку «Decimal View», чтобы все адреса представились в шестнадцатичном формате, и закрываем «Debugger». Теперь нам нужно дизассемблировать программу в виде последовательности байт, чтобы добавить данные к картинке в редакторе. В окне «ZX-Assembler++» переведите курсор под низ нашей программы. Нажмите кнопку «Disassemble». Откроется окно «Select Range To Disassemble». В параметре «From addr:» введите число 23755, а в «To addr:» - 23809. Установите галочку «As data block» и поставьте черную кнопку на пункт «DEFB»:

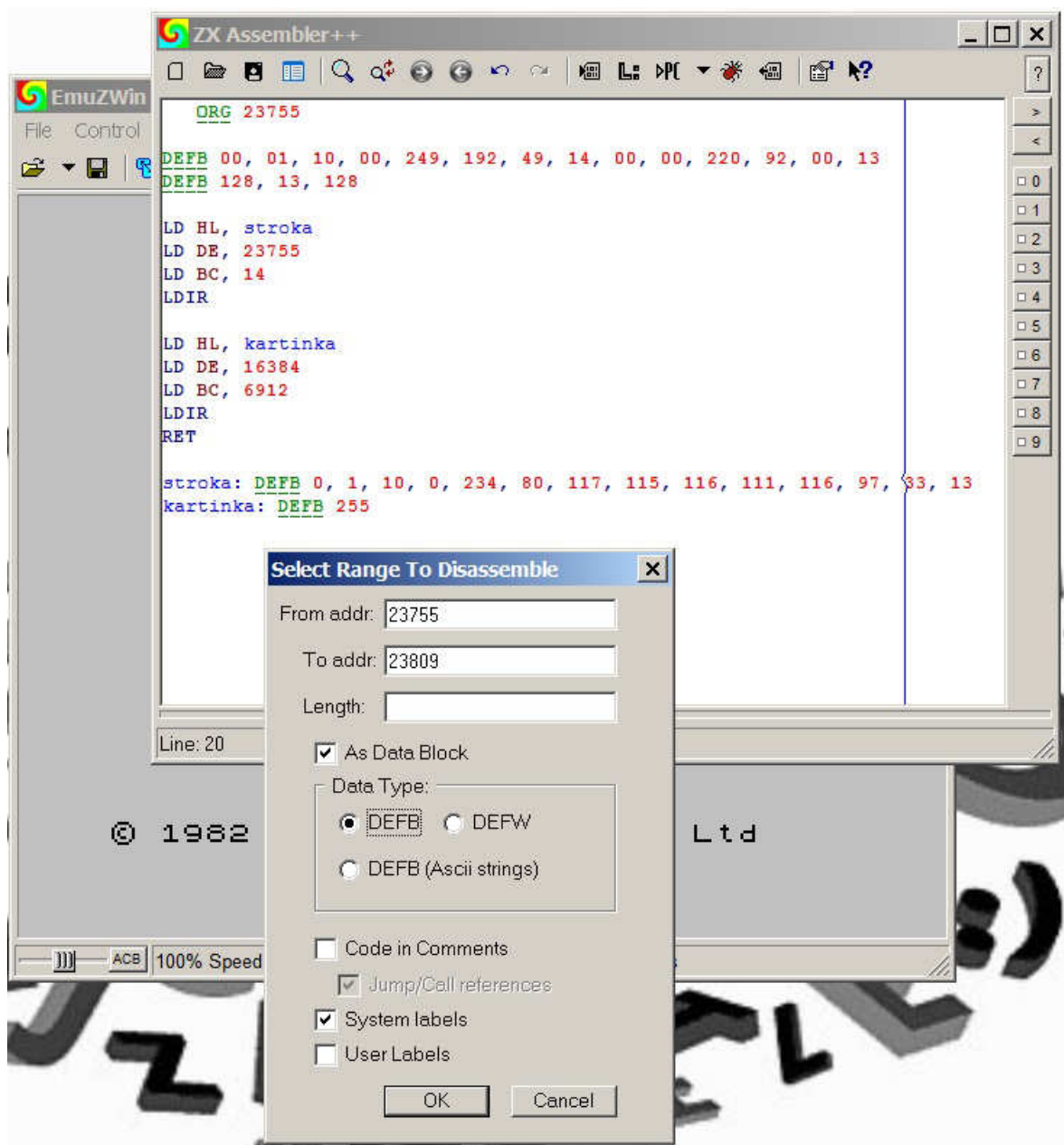


Рис. 5102. ZX-Assembler++. Установка параметров для дизассемблирования фрагмента данных.

Нажимаем «OK», и под текстом ассемблерной программы выдастся готовая цепочка байт нашей программы, в шестнадцатичном виде:

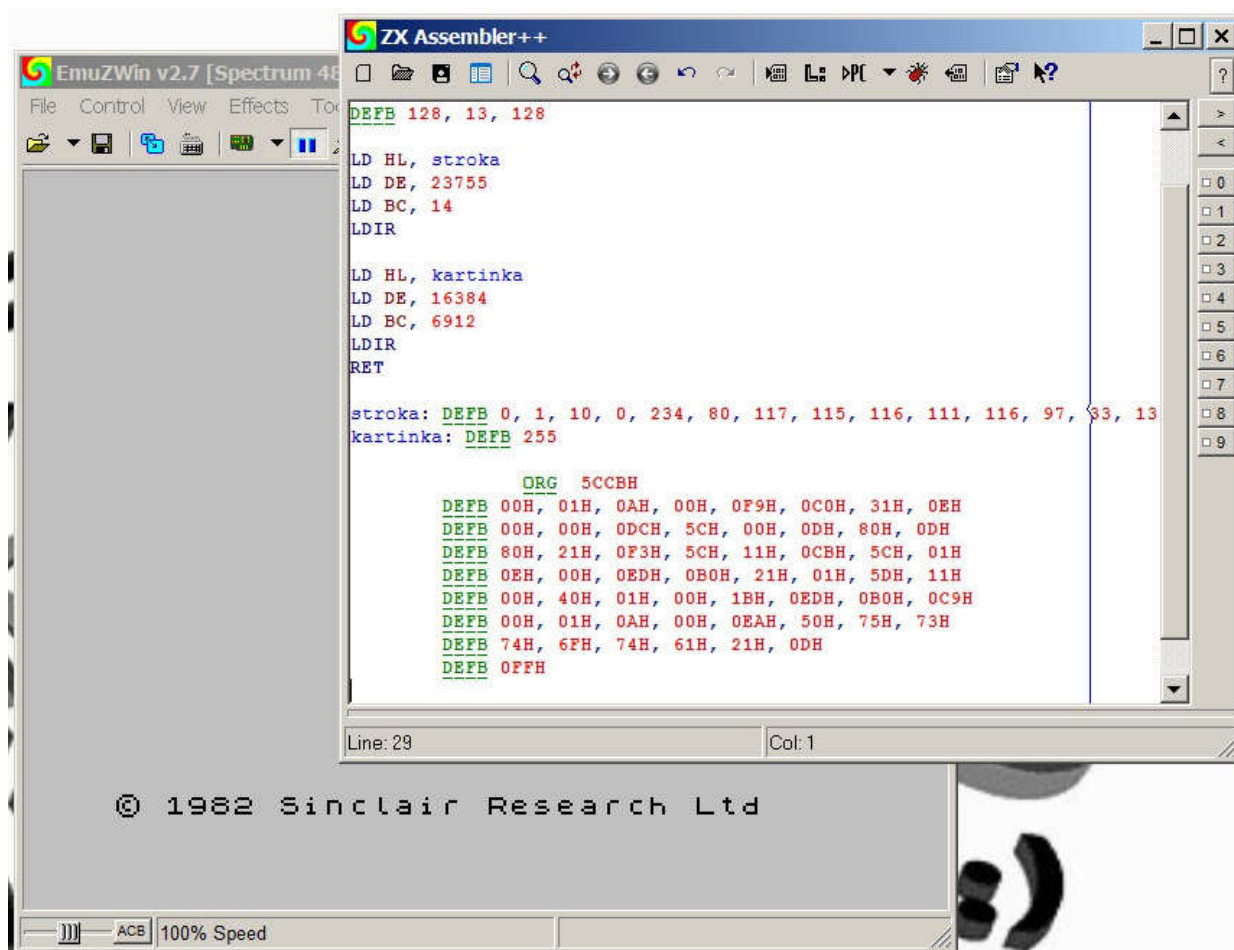


Рис. 5103. Преобразование программы в последовательность байт в шестнадцатичном виде.

Выделяем ее без ORG и нижней строки (она была для визуальной метки в отладчике), копируем в буфер обмена, и вставляем в текстовый редактор windows (блокнот, word, или любой другой удобный). Вся программа в кодах будет иметь размер 54 байта, и выглядеть так:

```

DEFB 00H, 01H, 0AH, 00H, 0F9H, 0C0H, 31H, 0EH
DEFB 00H, 00H, 0DCH, 5CH, 00H, 0DH, 80H, 0DH
DEFB 80H, 21H, 0F3H, 5CH, 11H, 0CBH, 5CH, 01H
DEFB 0EH, 00H, 0EDH, 0B0H, 21H, 01H, 5DH, 11H
DEFB 00H, 40H, 01H, 00H, 1BH, 0EDH, 0B0H, 0C9H
DEFB 00H, 01H, 0AH, 00H, 0EAH, 50H, 75H, 73H
DEFB 74H, 6FH, 74H, 61H, 21H, 0DH

```

Закрываем эмулятор. Теперь приведем ее в текстовом редакторе, к удобному виду, чтобы добавить в файл картинки редактором Hex Edit в виде последовательной цепочки шестнадцатичных значений. Для этого уберем все «H» в конце, пробелы и «DEFB». Программа станет вот такой:

```

00010A00F9C0310E0000DC5C000D800D8021F35C11CB5C010E00EDB021015D110
04001001BEDB0C900010A00EA507573746F7461210D

```

Подготовим картинку к модификации данных. Откроем Tapir, в нем картинку «test-scr.tap». Удалим заголовок «Bytes:» (клавиша DEL), и сохраним под временным именем «test-scr1.tzx».

Откроем Hedit, и только что измененный блок «test-scr1.tzx». Первым делом найдем служебную информацию, до данных, начинающихся с маркера «FF» (255) и удалим ее. Это будут первые 14 байт. Создадим еще один пустой файл (HexEd1 по умолчанию), и скопируем туда свою программу, преобразовав в шестнадцатичные данные, как описано в части 3 главе 5. Получится вот так:

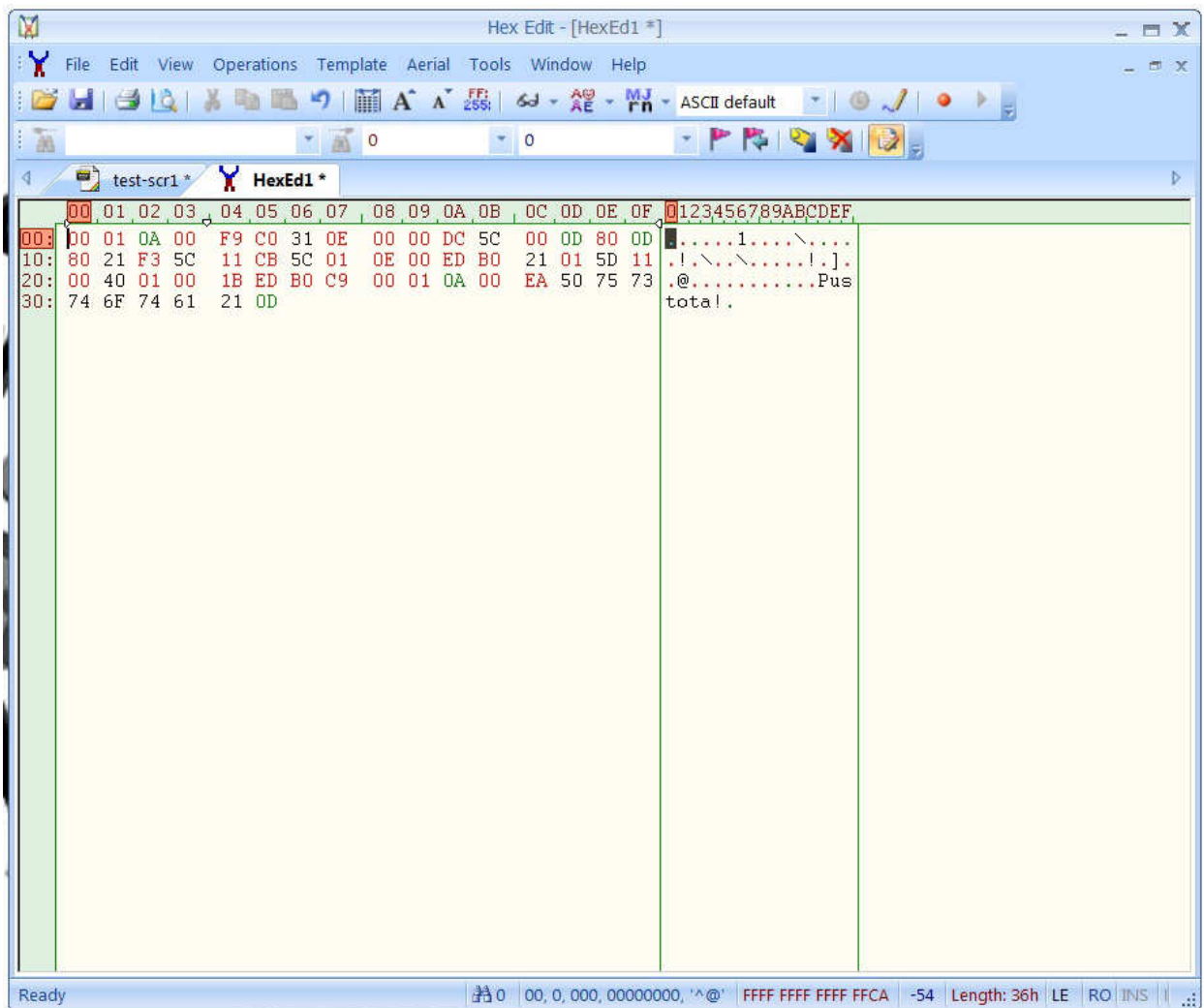


Рис. 5104. Hex Edit. Преобразование текстовых данных в последовательность шестнадцатичных байт.

Теперь скопируем в буфер получившийся код из 54 байт, и перейдем в окно с открытым «test-scr1.tzx». Установим курсор после «FF», но перед началом «80», и добавим данные в файл картинки, но не замещая байты, а дописывая в начало программы со смещением вниз. Для этого просто нажмем **CTRL+V**. В выскочившем сообщении будет предупреждение с выбором вставки данных:

«Pasting in overwrite mode will overwrite data! Do you want to turn on insert mode?»

В ответ нажимаем «Yes», и данные вклинятся туда, где стоял курсор, раздвинув строчки, и увеличив файл на 54 байта:

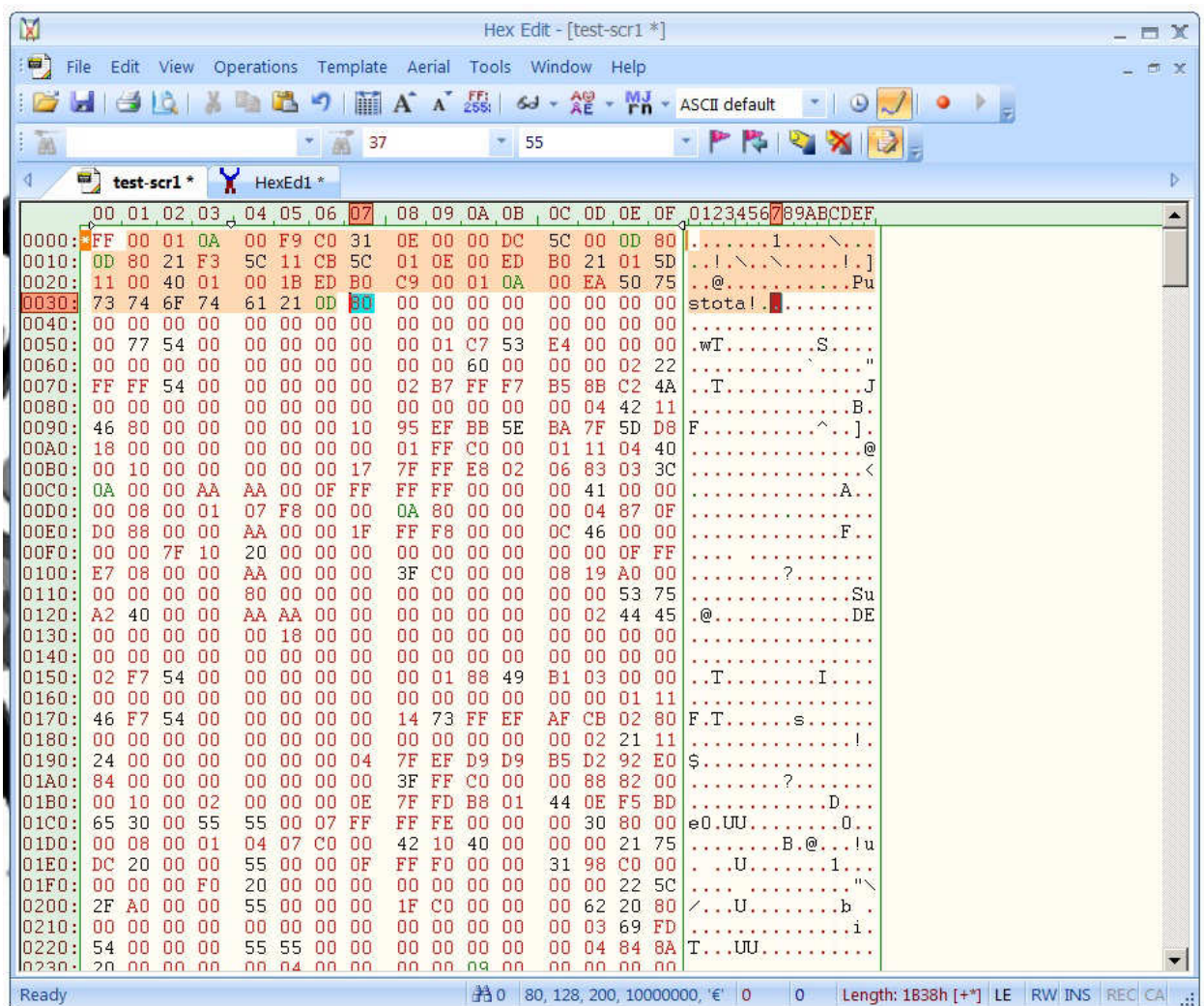


Рис. 5105. Модифицированный блок картинки со строкой BASIC и программой в машинных кодах.

Сохраните файл с картинкой «test-scr1.bin», а «HexEd1» закройте не сохраняя.

Снова откроем Tarig. Создадим новый блок кодов из данных файла «test-scr1.bin». Исправим в нем контрольную сумму на нужную, и добавим пилот-тон. (Подробно о создании блоков данных и заголовков говорилось в главе 5 четвертой части книги). В итоге у нас получился готовый блок данных для программы.

Теперь создадим заголовок. Программу назовем «progscri». Строку автостарта программы (Start) выставим 1, длина программы (Program length) будет соответствовать размеру строки 14 байт, а общая длина программы (Length) будет длина блока минус два служебных байта, то есть 6966:

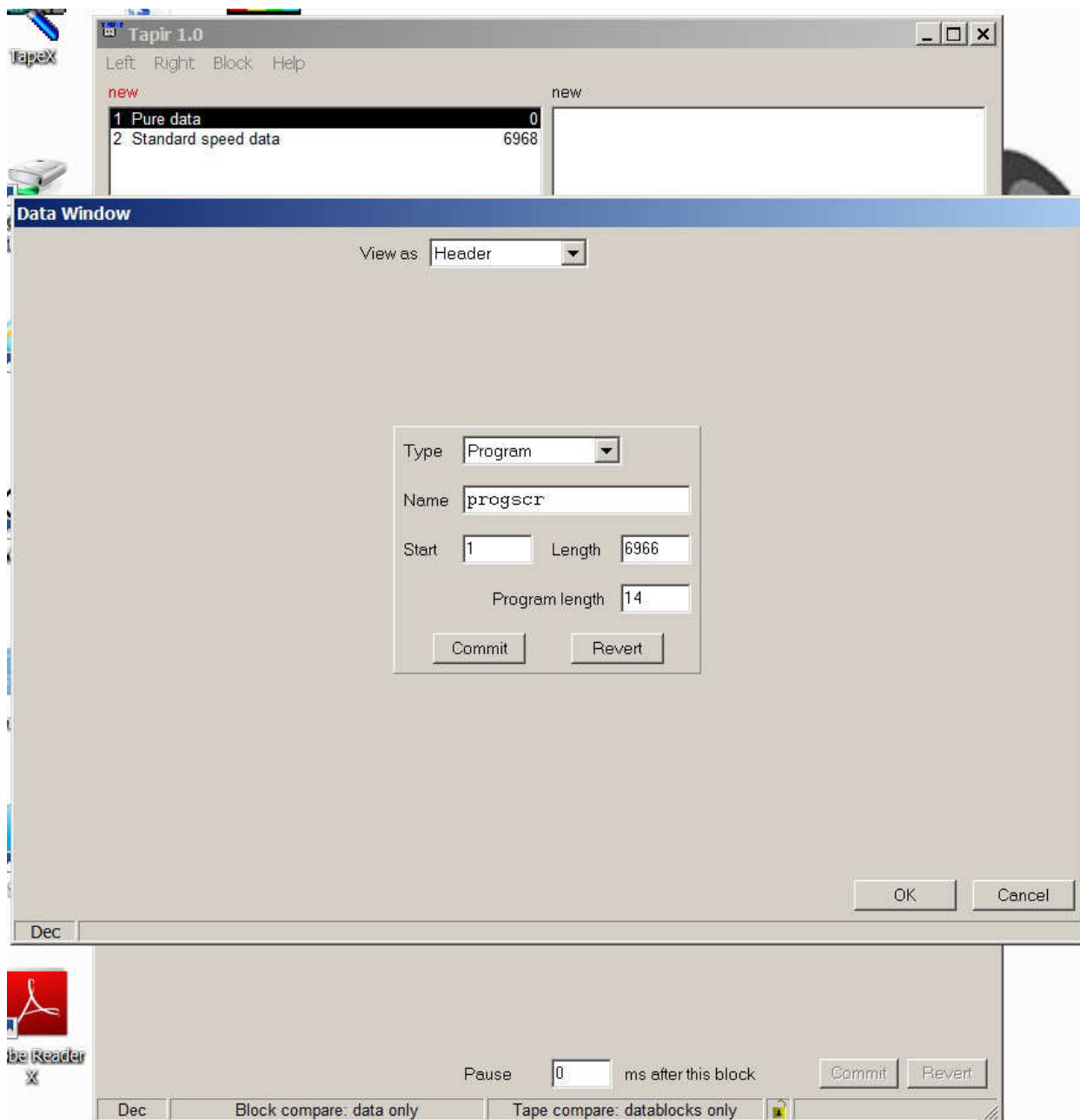


Рис. 5106. Tapir. Установка параметров заголовка «Program:» для гибридного блока данных.

Жмем «Commit», затем «OK». В главном окне устанавливаем промежуток между блоками 1000 миллисекунд, и добавляем пилот-тон. Устанавливаем заголовок перед блоком данных, сохраняем файл под именем «progscr.tzx», и закрываем программу.

Можно проверять работу созданной программы. В Spectaculator, после заголовка «Program:» звучат нехарактерные для такого блока звуки загружаемой картинки. После загрузки программа выведет картинку на экран и напишет сообщение:

⦿ OK, 1:1:

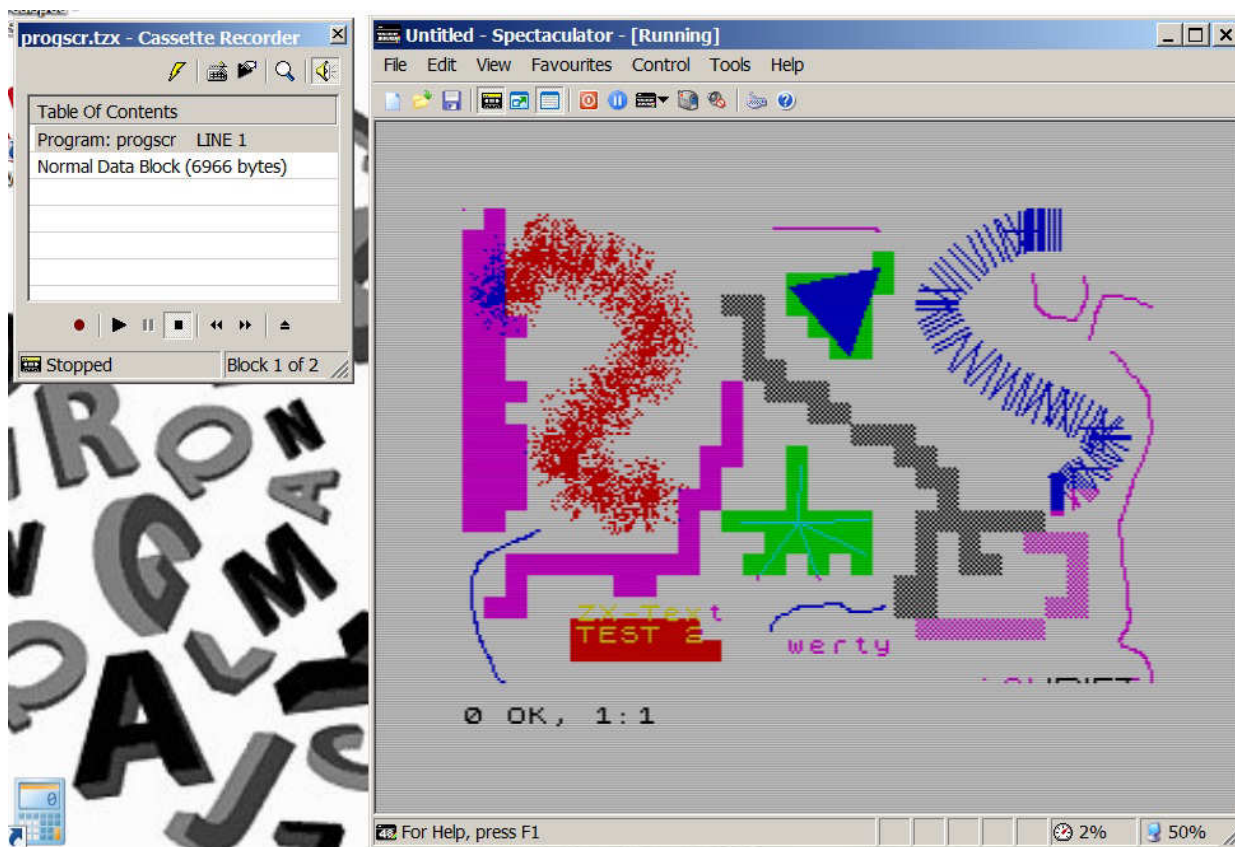


Рис. 5107. Spectaculator: Результат работы гибридного блока «Program:» с корректным выходом в BASIC.

Нажав «ENTER» и посмотрев программу, мы увидим единственную строку 1 REM Pustota!, которая корректно заместила строку автозапуска машинной программы.

Глава 2.

Загрузка BASIC программы блоком «Bytes:» и запуском через RANDOMIZE USR

Краткое содержание: разметка системных переменных, преобразование данных в BASIC программу, запуск и управление BASIC программой из машинных кодов, многократная подмена программ произвольно длины.

Мы уже пробовали писать программу с подменой строки. Но в том случае был упрощенный вариант, когда подменяемая строка была одной длины, и просто накрывала старую, не изменяя системных переменных. Давайте попробуем написать программу в «псевдо машинных кодах». Программа загружается в произвольную область памяти, а по команде RANDOMIZE USR ... формируется и запускается полноценная работоспособная BASIC программа. При этом требуется обеспечить корректный выход в интерпретатор, чтобы эту программу можно было редактировать, дописывать и запускать сколько угодно раз.

Откройте Spectaculator, посмотрите на данные системных переменных на пустом эмуляторе, сразу после сброса. Можете запомнить или записать значения ячеек, начиная с адреса 23755.

Теперь откройте любой файл с блоком «Program:», (можете открыть параллельно 2-й эмулятор), дайте загрузится заголовку, и выйдете в «Debugger» в паузу после заголовка, но перед загрузкой блока данных этой программы. Сравнив значения, вы увидите, что во время заголовка компьютер «подготовил» некоторые системные

переменные, вписав туда, в зависимости от размеров будущей программы определенные адреса. Он переразметил будущую область под BASIC, переменные, вводимую строку, и рабочую область.

Спектрум записывает программу на BASIC, если она запускалась длиной **PROG-E_LINE** (без маркера «128»). То есть вместе с областью **VARs** до маркера «128». Следует напомнить, что если вы хоть раз запускали для проверки свою программу, в которой задействован ввод данных, или работу с переменными, то в нее будут включены все переменные. Если эта программа не запускалась, то она запишется без учета переменных.

Например, имеется такая программа, которая затрагивает некоторые команды BASIC:

```
1 PRINT "test"
2 BEEP 1,0
3 GO SUB 10
4 FOR a=1 TO 10
5 BEEP 0.3,a
6 NEXT a
7 CIRCLE 125,87,80
8 GO TO 12
9 STOP
10 BORDER 6
11 RETURN
12 CLS
13 INPUT "Wwedi simwol";b$
14 PRINT PAPER 5;b$
```

Если вы запустите программу для проверки, и строке 13 наберете символы, то они запомнятся в переменной «b\$». Компьютер запомнит все это, и сохранит в блоке данных вместе с переменными.

Перейдем к написанию подготовительной программы, которая будет настраивать BASIC, перемещать и запускать программу. В общих чертах, алгоритм программы будет выглядеть следующим образом:

- 1) Разметка ОЗУ под новую BASIC программу.
- 2) Переброска программы BASIC в область обработки интерпретатором
- 3) Запуск BASIC программы из машинных кодов и выход.
- 4) Тело программы BASIC

Теперь все по порядку. Сначала выполним разметку программы. Для этого очистим область переменных путем запуска командой **RUN**, с номером строки, большим, чем имеется в программе. Например, введите **RUN 15**. После выдачи 0 OK, откроем «Debugger», и проанализируем что получилось.

Область **VARs** (23627/28) пустует, поэтому переменная будет указывать на адрес 23968 (160, 93) в котором стоит маркер «128». Переменная **E_LINE** (23641/42) укажет на ячейку 23969, в которой будет стоять значение «13» (ENTER).

При разметке, последними значимыми переменными будут: адрес рабочей области интерпретатора и стека калькулятора **WORKSP** (23649/50), **STKBOT** (23651/52) и **STKEND** (23653/54). Все три адреса указывают на ячейку 23971,(163, 93) памяти сразу за вторым маркером «128», в пустую область с мусором. Все остальное ОЗУ, во время старта программы, должно дополниться само. Дальше области пойдут расширяться, и смещать друг друга, как им вздумается.

Затем напишем программу переброски данных из произвольного адреса в 23755.

И наконец, переброшенную BASIC программу надо запустить из машинной программы, то есть совершить действие, обратное **RANDOMIZE USR ...**

Запуск BASIC из машинных кодов, имеет больше преимуществ, чем запуск BASIC программы из среды BASIC командами **RUN** и **GO TO**. Опирируя переменными **NEWPPS**

(23618/19) и **NSPPS** (23620) можно задать не только номер строки, которая сама запустится, но и выборочно номер оператора, с которого нужно запустить программу. В самом BASIC'е выполнить такое невозможно.

В конце BASIC программы в обязательном порядке прибавьте 128, 13, 128. Программа будет выглядеть так:

; разметка системных переменных BASIC

```
LD HL, 23968
LD (23627), HL
LD HL, 23969
LD (23641), HL
LD HL, 23971
LD (23649), HL
LD (23651), HL
LD (23653), HL
```

; переброска и формирование BASIC программы

```
LD HL, programa
LD DE, 23755
LD BC, 216
LDIR
```

; запуск BASIC из машинных кодов

```
LD A, 1
LD (23618), A ; выполнить с 1-й строки BASIC
LD A, 1
LD (23620), A ; выполнить с 1-го оператора в 1-й строке BASIC
RET
```

programa: DEFB

Преобразуем BASIC программу в блок данных. Откройте EmulZWin. Набрав программу, запустите ее, и посмотрите, что получится. После проверки работоспособности, восстановите рамку в белый цвет и наберите **FCIM 15**, чтобы очистилась вся область переменных, которая снова сожмется до 1 байта. В «*Debugger'e*» проверьте еще раз размер программы путем вычитания **WORKSP** из **PROG**. Получится 216 байт.

Дизассемблируйте всю программу в виде строки с десятичными данными DEFB с адреса 23755 по 23970. У вас получится такая длинная программа:

```
DEFB 0, 1, 8, 0, 245, 34, 116, 101
DEFB 115, 116, 34, 13, 0, 2, 17, 0
DEFB 215, 49, 14, 0, 0, 1, 0, 0
DEFB 44, 48, 14, 0, 0, 0, 0, 0
DEFB 13, 0, 3, 10, 0, 237, 49, 48
DEFB 14, 0, 0, 10, 0, 0, 13, 0
DEFB 4, 20, 0, 235, 97, 61, 49, 14
DEFB 0, 0, 1, 0, 0, 204, 49, 48
DEFB 14, 0, 0, 10, 0, 0, 13, 0
```

```

DEFB 5, 13, 0, 215, 48, 46, 51, 14
DEFB 127, 25, 153, 153, 153, 44, 97, 13
DEFB 0, 6, 3, 0, 243, 97, 13, 0
DEFB 7, 29, 0, 216, 49, 50, 53, 14
DEFB 0, 0, 125, 0, 0, 44, 56, 55
DEFB 14, 0, 0, 87, 0, 0, 44, 56
DEFB 48, 14, 0, 0, 80, 0, 0, 13
DEFB 0, 8, 10, 0, 236, 49, 50, 14
DEFB 0, 0, 12, 0, 0, 13, 0, 9
DEFB 2, 0, 226, 13, 0, 10, 9, 0
DEFB 231, 54, 14, 0, 0, 6, 0, 0
DEFB 13, 0, 11, 2, 0, 254, 13, 0
DEFB 12, 2, 0, 251, 13, 0, 13, 19
DEFB 0, 238, 34, 87, 119, 101, 100, 105
DEFB 32, 115, 105, 109, 119, 111, 108, 34
DEFB 59, 98, 36, 13, 0, 14, 13, 0
DEFB 245, 218, 53, 14, 0, 0, 5, 0
DEFB 0, 59, 98, 36, 13, 128, 13, 128

```

Вставьте эту программу, вместо DEFB за меткой «programa», после созданной программы-загрузчика. Перед началом всей программы установите ORG 40000.

Перезагрузим эмулятор, скомпилируем программу, и через окно «Debugger» определим ее окончательную длину, которая получится 262 байта. В итоге должно получиться, как на картинке:

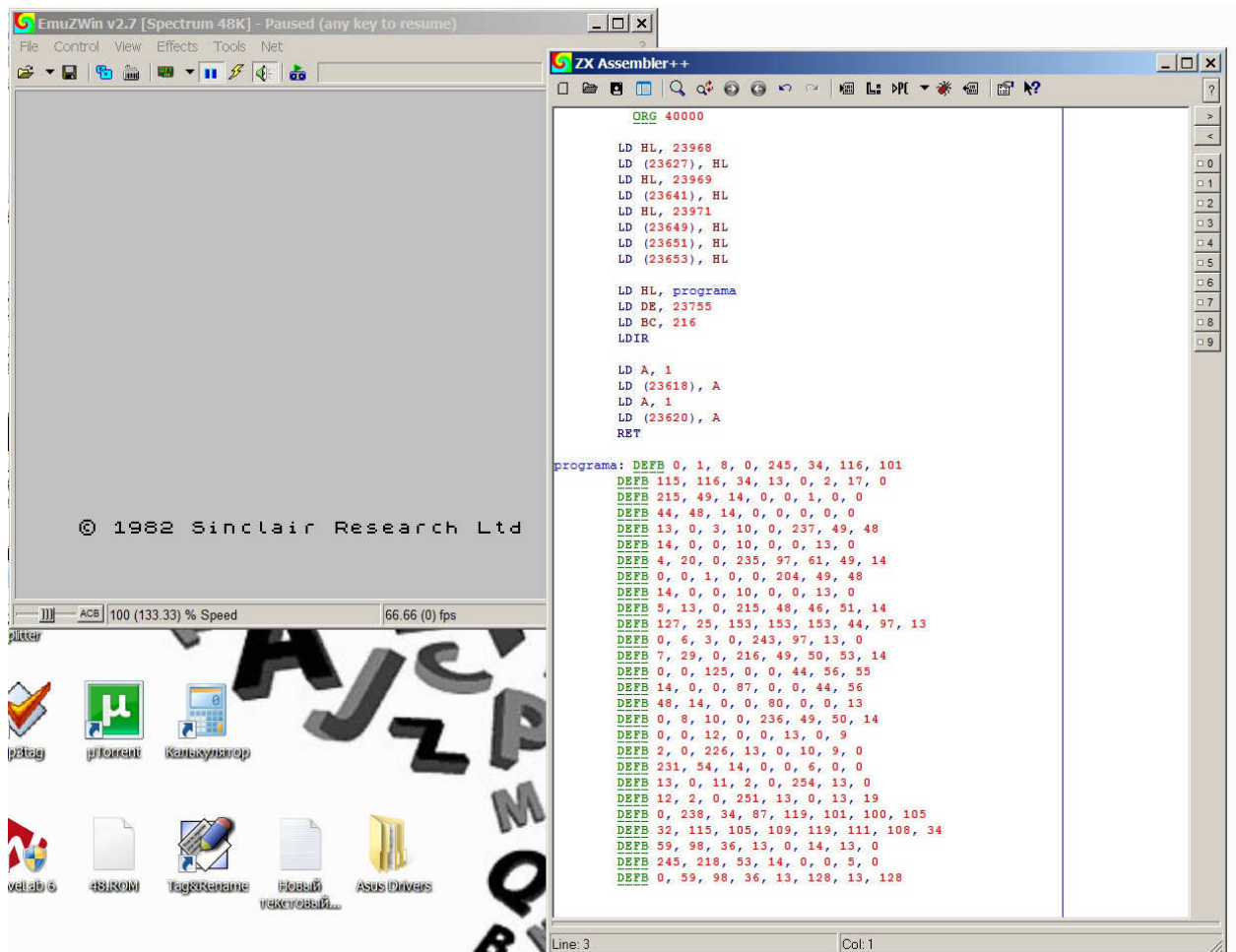


Рис. 5200. Окно Assembler++: Окончательно созданная программа на ассемблере.

Перез тем, как ее записать, наберите на BASIC программу-загрузчик:

```
1 LOAD ""CODE
```

Теперь, сделайте строку нулевой (`POKE 23756, 0`)
Затем введите:

```
1 RANDOMIZE USR 40000
```

Теперь наберем в нижней строке подготовительную программу для записи *.tzx файла:

```
SAVE "MAGICBASIC" LINE 0: SAVE  
"magicbasic"CODE 40000,262
```

Сохраним эту программу под именем «*magicbasic.z80*»:

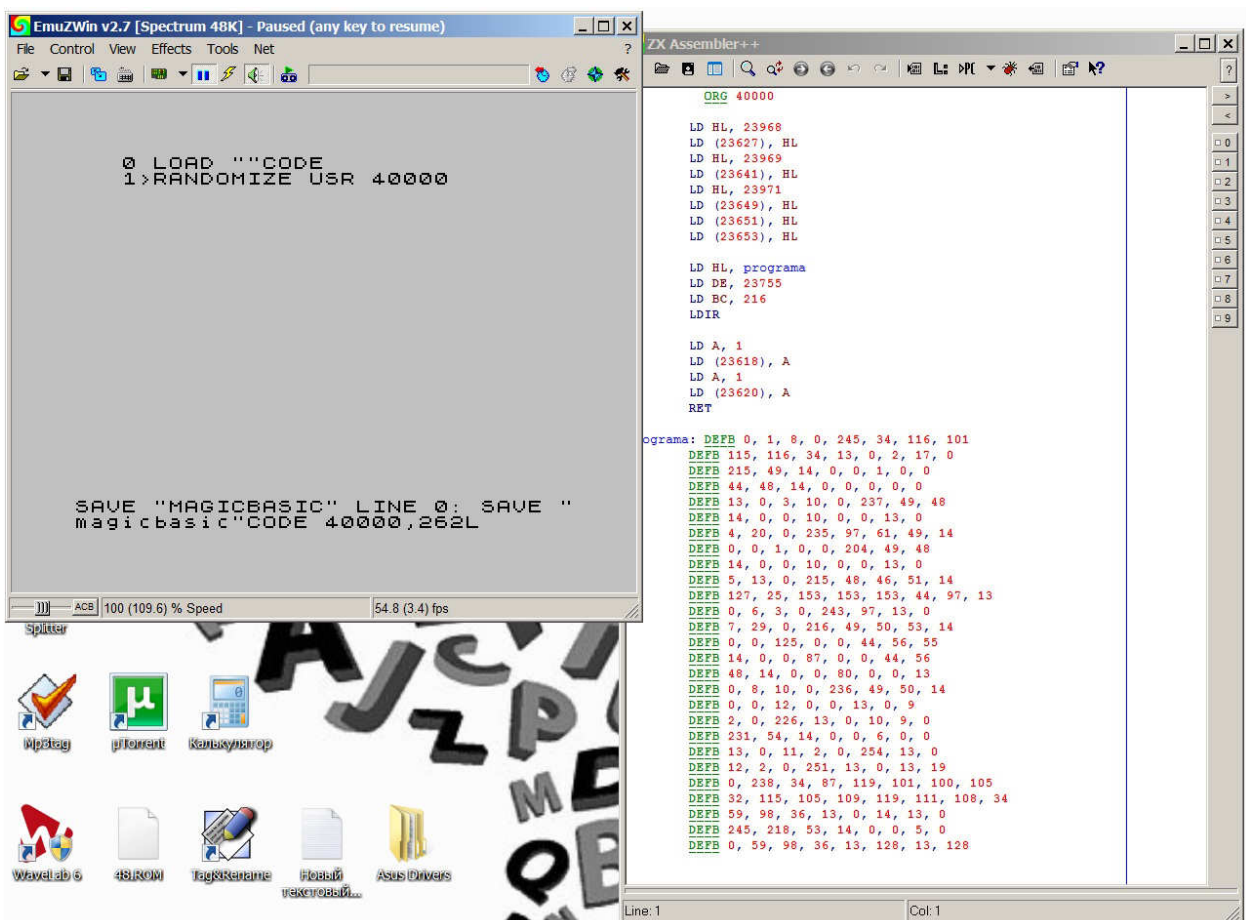


Рис. 5201. EmuZWin: Подготовленная программа для записи многоблокового *.tzx файла.

Закрываем эмулятор, и создаем 2-х блоковый файл в Realspectrum, под именем «*magicbasic.tzx*». Затем проверяем в Spectaculator по `LOAD ""ENTER`.

После загрузки обоих блоков, машинная программа стартует, размечает область, переносит туда новую BASIC программу, затирая старую, а затем настраивает ее запуск с указанной строки, после чего успешно выходит по команде `RET`. Интерпретатор тут же подхватывает измененные данные переменных **NEWPPS** и **NSPPS**, и немедленно запускает созданную BASIC-программу:

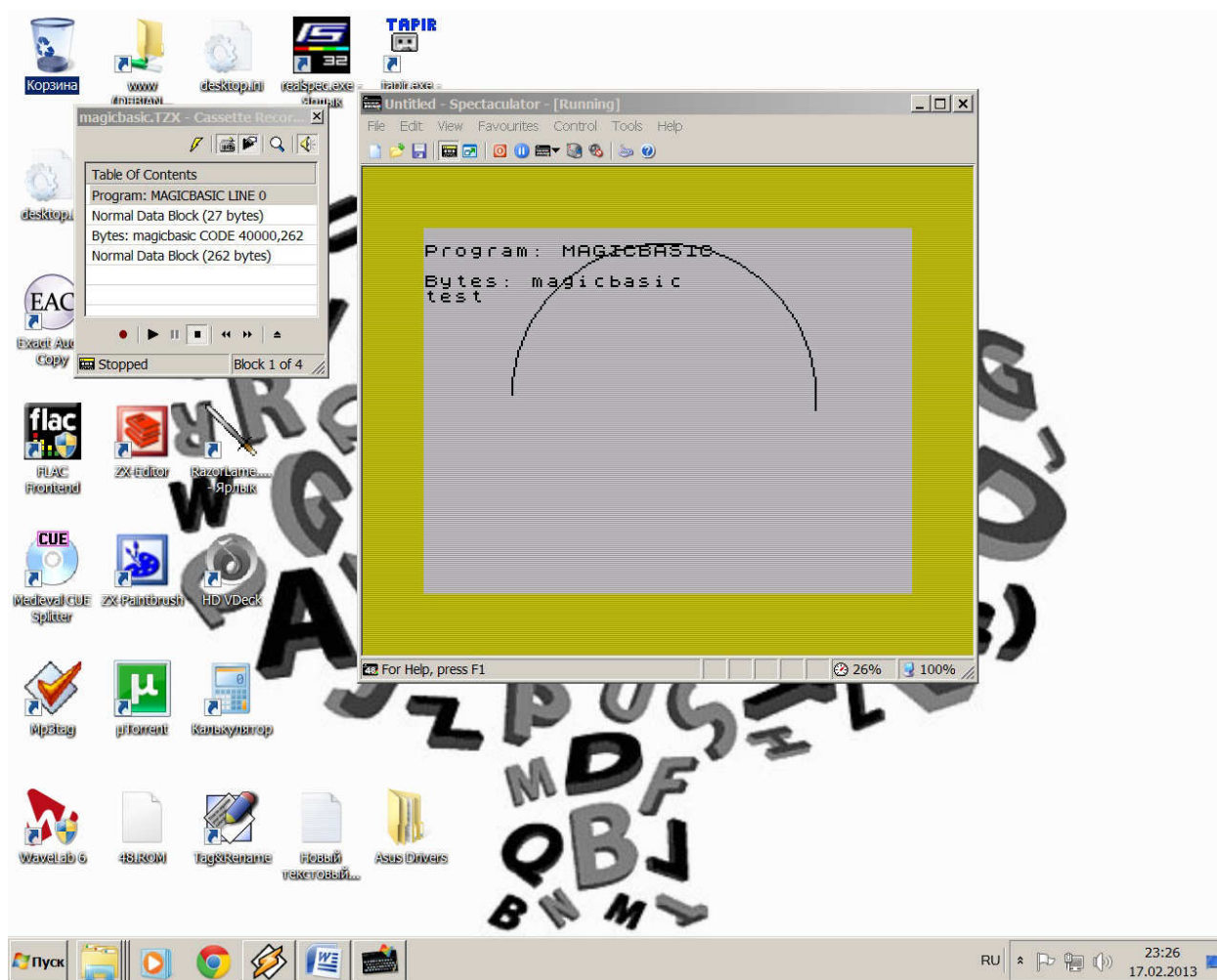


Рис. 5202. Проверка выполнения сформированной BASIC программы после загрузки и переброски.

После того как проигрывает мелодия, рамка станет желтой и экран очистится. В нижней строке компьютер попросит ввести любые символы. Можете набрать любые несколько символов, они немедленно напечатаются на экране, и программа благополучно завершится.

Нажав **ЕНТЕР**, увидим, что от программы-загрузчика с нулевой строкой, не осталось и следа, а на экране совершенно другая программа, большего размера, которая корректно работает:

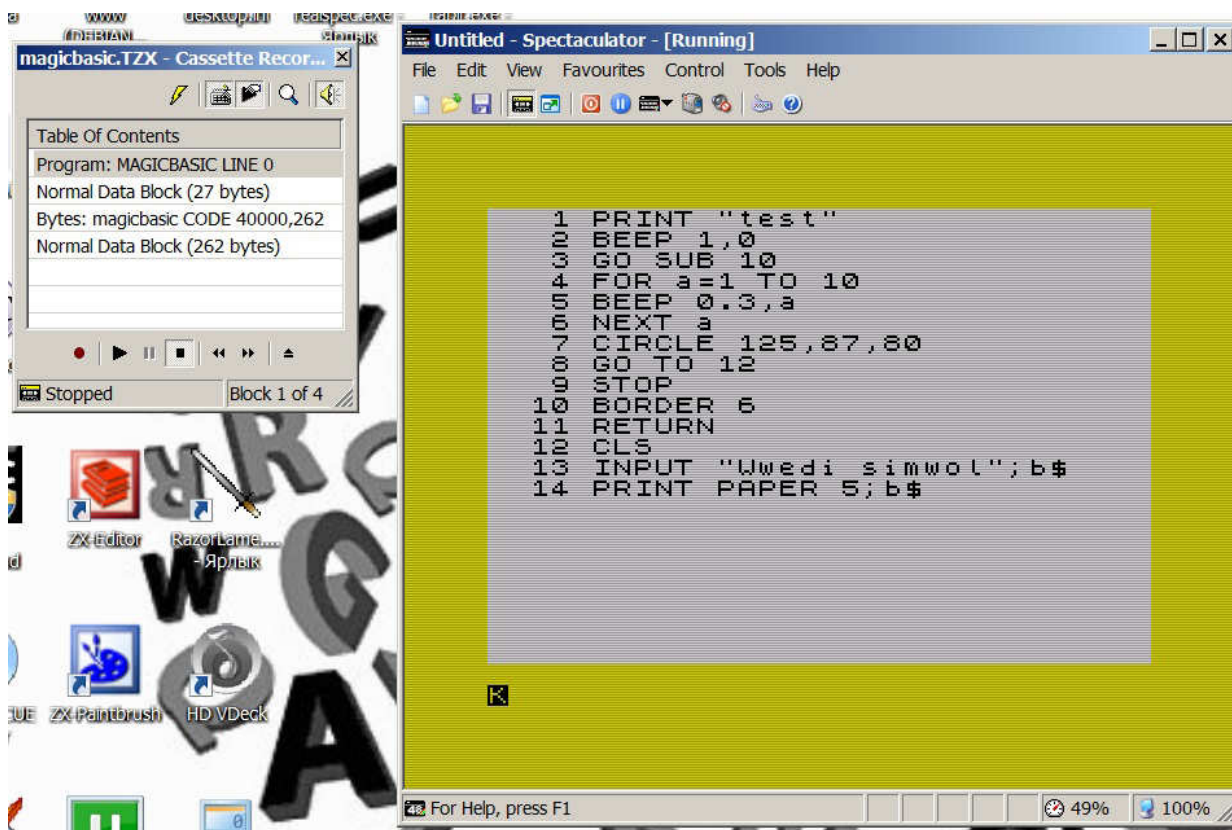


Рис. 5203. Spectaculator: Сформированные работоспособные строки BASIC программы.

Вы можете изменять или стирать строки, но набрав `RANDOMIZE USR 40000`, программа восстановится, появившись целой и невредимой.

Упражнение:

Доработайте данную программу так, чтобы в памяти держать одновременно три программы на BASIC, разной длины и сложности, а запуск этих программ сделать, по `RANDOMIZE USR 40000`, `45000` и `50000`.

Глава 3. ZX-DESKTOP, или ностальгия по windows.

Краткое содержание: параллельная работа BASIC и машинных кодов, режим прерываний IM2, заставка «рабочего стола».

Давайте создадим фоновый рисунок рабочего стола для спектрума, с возможностью подгрузки собственных картинок. При этом узоры на экране должны сохраняться во время параллельной работы программ на BASIC, включая нижние строки. При этом постараемся максимально автоматизировать процесс загрузки, и запуска, чтобы нашей задачей было набрать `LOAD ""` и больше не беспокоится. Потом останется только подгружать собственные картинки для рабочего стола.

Вспомним методы автозагрузки, рассматриваемые во 2-й части книги. Так как программа не длинная, ее можно запихать в нижнюю часть памяти, чтобы была возможность загружать в верхние адреса сразу несколько картинок. Дополнить программу можно автозагрузкой через машинный стек, с последующим его восстановлением, и работоспособностью интерпретатора BASIC. Это также освободит от лишних манипуляций с запуском «активатора» прерываний режима IM 2.

Откройте эмулятор EmulzWin, и первым делом наберите `CLEAR 29999`, потому что предстоит опасная работа со стекком. Далее откроем Ассемблер и введем (или скопируем через буфер обмена) следующую программу. После символа точка с запятой

идут комментарии с разъяснениями работы программы. При вставке в ассемблер, хоть они и не должны влиять на компиляцию, все равно рекомендую убрать:

ORG 65362

DEFB 88, 255 ; новый адрес возврата из стека, указывает на нашу программу
DEFB 3, 19, 0, 62 ; текущие значения стека, замеченные во время загрузки

; блок переброски параллельной программы фона в адрес ее параллельного выполнения.

LD HL, paral
LD DE, 60208
LD BC, 23
LDIR

; установим желто-зеленый линолеум в качестве примера фона. Его потом можно менять на собственные рисунки, загружаемые в адрес 30000, которые будут появляться.

LD HL, 30000
LD D, 0
cicl: LD B, 16
polos1: LD A, 40
LD (HL), A
INC HL
LD A, 48
LD (HL), A
INC HL
DJNZ polos1
LD B, 16
polos2: LD A, 48
LD (HL), A
INC HL
LD A, 40
LD (HL), A
INC HL
DJNZ polos2
INC D
LD A, D
CP 12
JR NZ, cicl

; активация параллельного процесса фоновой заставки, с помощью режима IM 2. В
; регистр I запишем 48, следовательно параллельную программу поместим в адрес 60208.
; Программа одноразовая, и больше ее запускать не нужно.

DI
LD A, 48
LD I, A
IM 2
EI
RET

; программа фоновой заставки, которая перебрасывается в адрес 60208, где будет ; параллельно выполняться. После переброски в этом месте больше не нужна.

```
paral: DI
      PUSH AF
      PUSH HL
      PUSH DE
      PUSH BC

      LD HL, 30000
      LD DE, 22528
      LD BC, 768
      LDIR

      POP BC
      POP DE
      POP HL
      POP AF
      JP 56      ; выход в BASIC
```

Скомпилировав в память EmulZWin, откроем отладчик, и рассчитаем длину программы. Она должна получится 84 байта. Записать ее можно двумя вариантами. Отдельным файлом «*Bytes :*» или классическим многоблочным файлом, «*Program - Bytes*», где сначала загружается программа, которая загружает «*Bytes :*». Можно сделать оба варианта.

Для первого варианта в нижней строке эмулятора просто набираем:

```
SAVE "zx-desktop"CODE 65362,84
```

Затем сохраняем под именем «*zx-desktop.z80*», и создаем *.tap / *.tzx файл. Загружать программу будем по `LOAD ""CODE`.

В качестве примера, подробно опишу вариант №2. Вместе с загрузчиком, состоящим из одной строки, перед сохранением, напишем BASIC строку с памяткой:

```
1 LOAD ""CODE
2 REM Podgruzka cwetnyh atributow dlq fona
komandoj LOAD ""CODE 30000 ili izmenitx w
realxnom wremeni POKE 30000, xxx
```

На экране это будет выглядеть так:

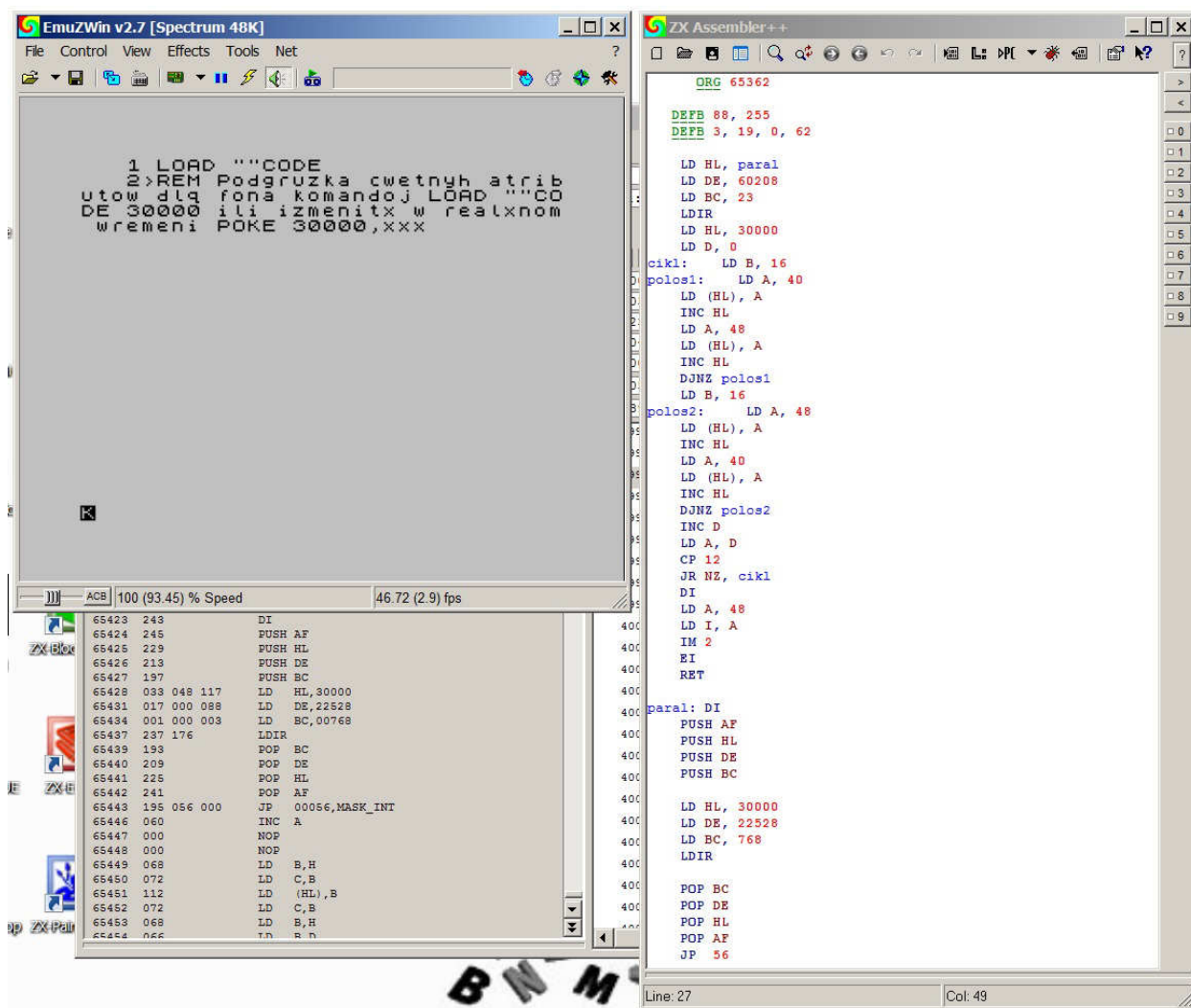


Рис. 5300. Готовые программы на BASIC'e и ассемблере, перед записью.

После ввода программы в нижней строке напишем команды записи на 2 блока:

```

SAVE "ZX-DESKTOP" LINE 1: SAVE "zx-
desktop"CODE 65362,84

```

Сохраните файл под именем «zx-desktop1.z80», а в Realspectrum создайте 2-х блоковый *.tap / *.tzx файл.

При проверке работоспособности на Spectaculator, после загрузки программа должна автоматически стартовать, окрасив экран желто-зеленым линолеумом. В клетках, на фоне него, будут чернеть строки BASIC программы:

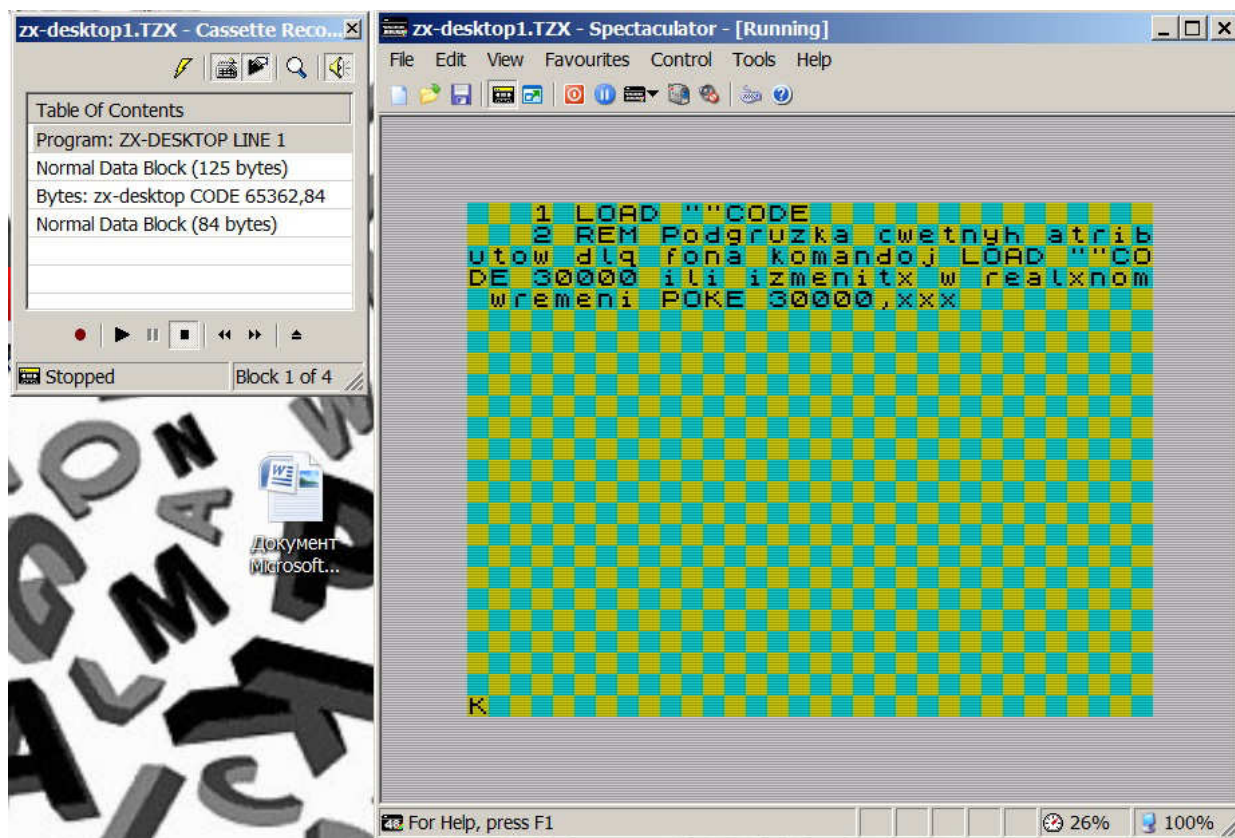


Рис. 5301. Параллельная работа интерпретатора BASIC и вывода картинки на экран в режиме IM 2.

Вы можете попробовать нарисовать кружочек, подав команду `CIRCLE 125, 88, 80`. Во время выполнения этой команды все равно будет присутствовать фон:

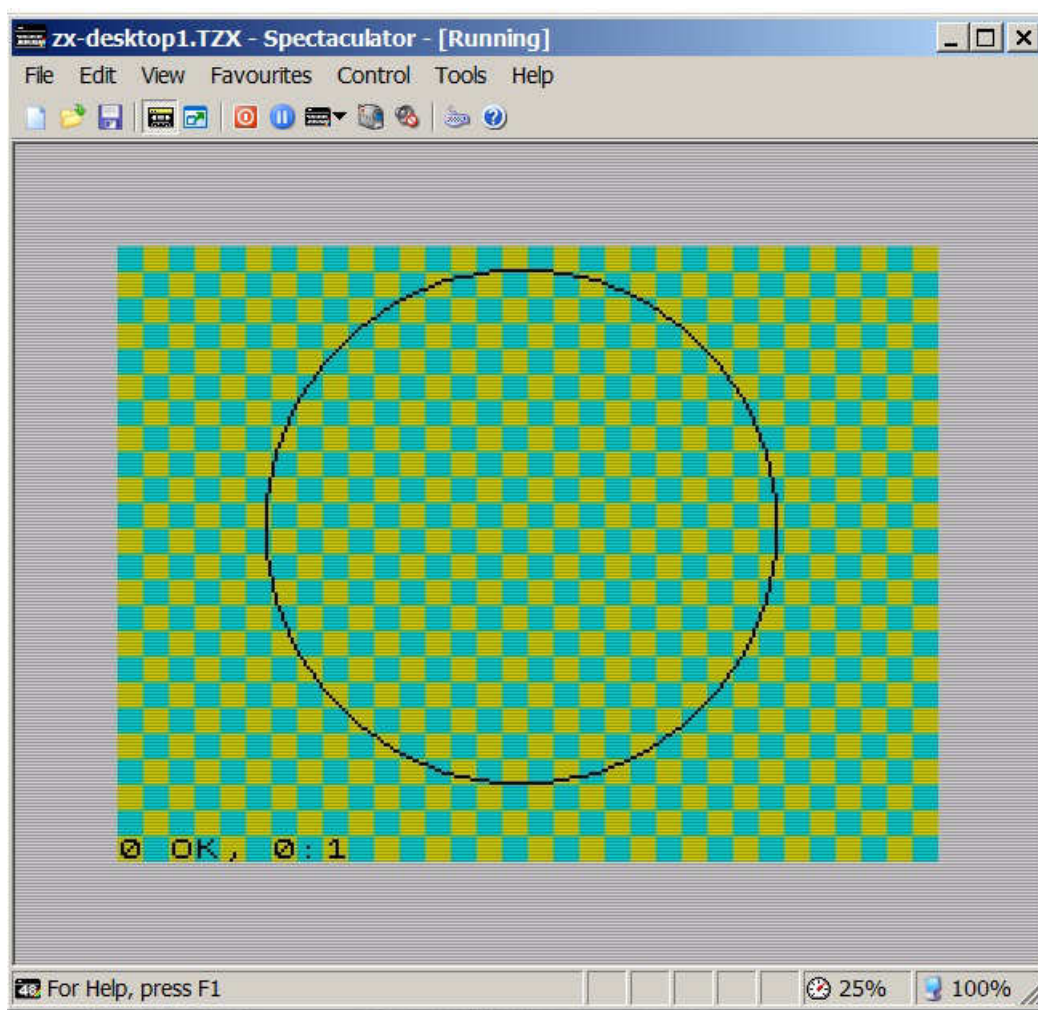


Рис. 5302. Рисование кружочка по «фону рабочего стола».

Самая ценная функция, это то, что программы работают действительно параллельно. Единственным условием для этого являются сохранение всех регистров, после выполнения собственной параллельной программы. Иначе может все перекосять, компьютер зависнет, или перезагрузится. Недостаток программы – невозможность одновременно задать цвета в BASIC программе. Компьютер их просто затирает фоновым рисунком.

Глава 4.

Создание картинки заголовками с коротким пилот-тоном и сохранением в память.

Краткое содержание: создание графики в имени заголовка, выборочная выемка данных, сборка нестандартных блоков, работа с параметрами turbo speed, изменение длины пилот-тона, фотографирование текущего состояния экрана.

В главе 8 второй части книги, мы уже пробовали создавать заголовок с управляющими символами. А теперь вспомним такую ситуацию, в которую многие хоть раз, но попадали. Некоторые программы, вместо блока «Program:», начинаются с «Bytes:». Машинально набрав LOAD "", выскакивает заголовок «Bytes:», а тело программы игнорируется. Затем считывается следующий заголовок, и выбивается на строчку ниже первого. Если заголовок снова не «Program:», то блок данных опять игнорируется. Так будет продолжаться, пока компьютер не встретит нужный ему тип

заголовка. Никогда не приходила мысль использовать это свойство, например, для создания рисунка на экране?

Давайте попробуем создать программу, которая загружается по `LOAD ""ENTER,` использующая это свойство, и рисующая заголовками, картинку, а потом сохраняющая ее в произвольную область памяти. Пусть это будет простой смайлик. Для его прорисовки будем использовать 10 байт, отведенные для названия блока заголовка. Из них оставим 3 символа на управляющую команду АТ с координатами. В итоге, на горизонтальную полосу рисунка, остается 7 байт.

Вначале спроектируем и нарисуем картинку смайлика. Будем рисовать его графическими символами в режиме курсора «Б». Откроем EmulZwin и наберем программу на BASIC, помогающую вычислить графические коды. Она будет выглядеть так:

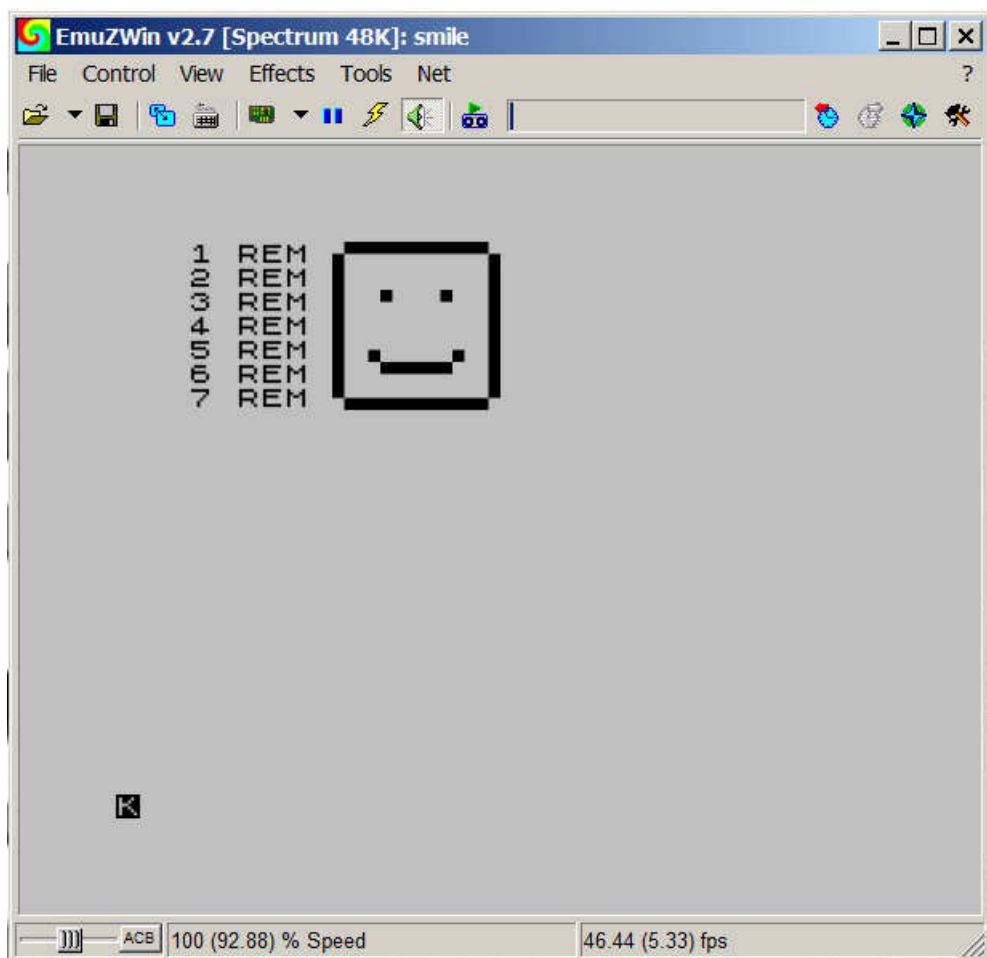


Рис. 5400. Рисунок смайлика в виде последовательных строк на BASIC в команде REM.

Программа займет 91 байт памяти, с адреса 23755. Откроем окно «Assembler++» и дизассемблируем программу, отбросим техническую информацию и коды команды REM. У нас получится следующие строки, с кодами графических символов, рисующие смайлик:

```
DEFB 137, 131, 131, 131, 131, 131, 134
DEFB 138, 32, 32, 32, 32, 32, 133
DEFB 138, 32, 130, 32, 129, 32, 133
DEFB 138, 32, 32, 32, 32, 32, 133
DEFB 138, 132, 32, 32, 32, 136, 133
DEFB 138, 32, 131, 131, 131, 32, 133
DEFB 134, 140, 140, 140, 140, 140, 137
```

Так как картинка будет иметь фиксированные координаты, рисуясь с левого верхнего угла, то добавим к каждой строке управляющий код «22» (команду AT) с двумя координатами после нее. В итоге получатся готовые 10-ти байтные имена для семи заголовков:

```
DEFB 22, 0, 0, 137, 131, 131, 131, 131, 131, 134
DEFB 22, 1, 0, 138, 32, 32, 32, 32, 32, 133
DEFB 22, 2, 0, 138, 32, 130, 32, 129, 32, 133
DEFB 22, 3, 0, 138, 32, 32, 32, 32, 32, 133
DEFB 22, 4, 0, 138, 132, 32, 32, 32, 136, 133
DEFB 22, 5, 0, 138, 32, 131, 131, 131, 32, 133
DEFB 22, 6, 0, 134, 140, 140, 140, 140, 140, 137
```

А теперь подумаем, какая будет структура и последовательность блоков создаваемого сложного *.tzh файла:

1. Заголовок «Bytes:» с 1-й строкой рисунка
2. Заголовок «Bytes:» со 2-й строкой рисунка
3. Заголовок «Bytes:» с 3-й строкой рисунка
4. Заголовок «Bytes:» с 4-й строкой рисунка
5. Заголовок «Bytes:» с 5-й строкой рисунка
6. Заголовок «Bytes:» с 6-й строкой рисунка
7. Заголовок «Program:» с 7-й строкой рисунка
8. Данные блока «Program:» с BASIC строкой
9. Заголовок «Bytes:» с самозатирающейся строкой и автостартом из стека
10. Данные блока «Bytes:» с программой переносом данных из 16384 в 30000

Первым делом создадим технические автономные блоки программ (пункты 7, 8, 9 и 10), а потом все соберем в единый блок программой Tapir, расставив в нужной последовательности. Создадим блок «Program:», который будет удовлетворять условию LOAD "", завершая бесконечный цикл считывания заголовков. Откроем Realspectrum и наберем такую строку:

```
1 LOAD "sml-loader" CODE: REM RANDOMIZE USR
30000
```

Команду RANDOMIZE, стоящую после REM, следует набрать не побуквенно, а именно командой, чтобы сохранить длину блока, в районе 19 байт, замаскировав под заголовок (например, через THEN, с последующим ее затиранием, или добавлением REM после печати RANDOMIZE). Затем наберите в нижней строке подготовительную программу, для записи блока:

```
SAVE "sml-loader" LINE 1
```

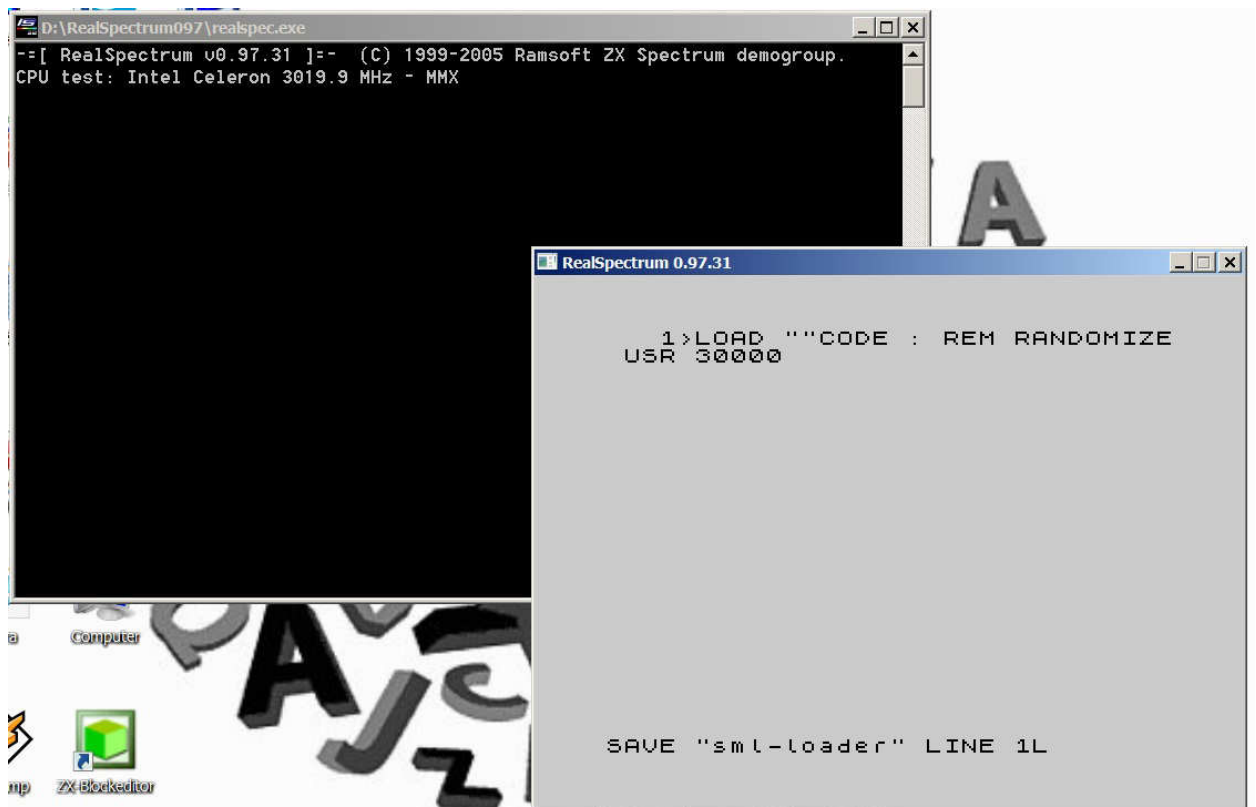


Рис. 5401. Создание блока-загрузчика «Program:» для окончания действия команды `LOAD""`.

Сохраним в *.z80, и создадим файл *.tzx под одноименным названием «*sml-loader.tzx*». Теперь создадим автозапускной блок кодов, с переносом картинки в память, и программы обратной переброски в область экрана. Это будет автозапускная программа-фотограф, которая запоминает текущее состояние экрана, и перебрасывает его в свободную область памяти. Программа будет следующей:

```
ORG 65362
```

```
DEFB 88, 255
DEFB 3, 19, 0, 62
```

```
LD HL, TEXT
LD DE, 30000
LD BC, 12
LDIR
```

```
LD HL, 16384
LD DE, 30012
LD BC, 6912
LDIR
JP 7030
```

```
TEXT: LD HL, 30012
      LD DE, 16384
      LD BC, 6912
      LDIR
      RET
```

Откройте EmuZWin. Наберите `CLEAR 39999`, а затем скопируйте в окно «ZX-Assembler++» эту программу, и скомпилируйте ее. Программа займет в памяти 43 байта. В нижней строке наберите:

```
SAVE "photo-pict"CODE 65362,43 :
```

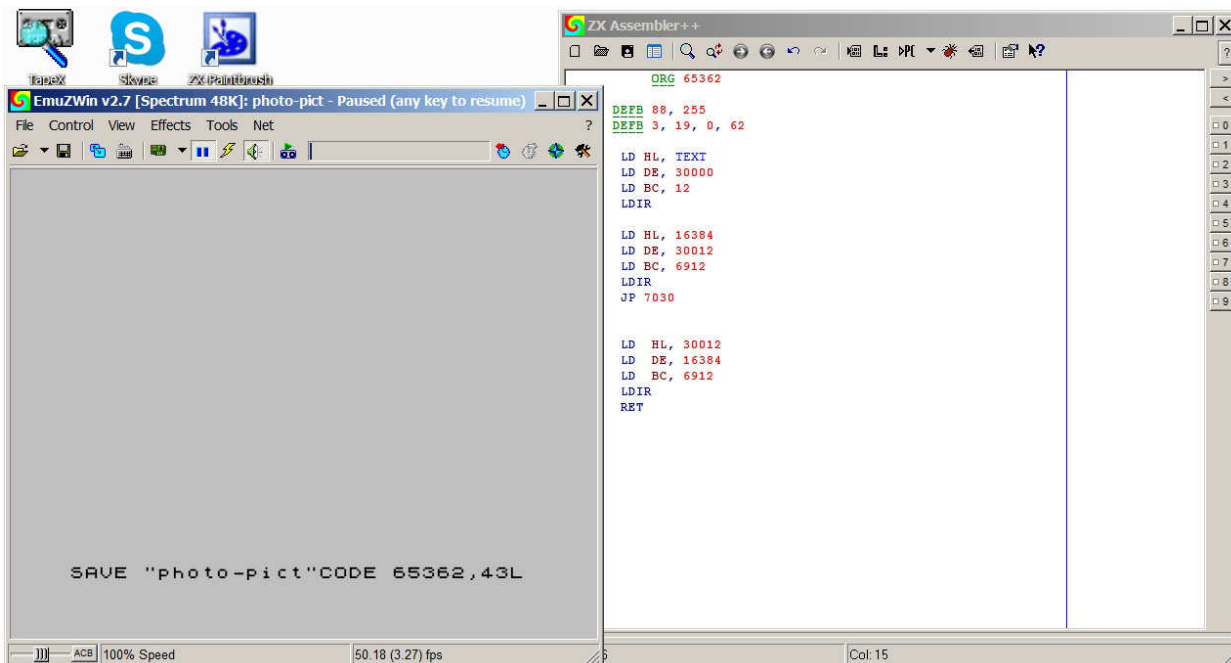


Рис. 5402. Подготовка к записи автостартующей программы, фотографирующей состояние экрана.

Сохраните под одноименным названием файл «*photo-pict.z80*». Создайте «*photo-pict.tzx*» в Realspectrum. Можете проверить ее работоспособность в Spectaculator'е, набрав для загрузки в нижней строке: `PRINT "аааааааа": LOAD ""CODE`.

Теперь приступаем к самой сложной части. В Tapir Создадим файл *.tzx, состоящий из шести заголовков подряд. Сначала создадим шаблон одного заголовка, по методу, рассматриваемому в главе 6 части 4. Сохраним файл-заголовок, записав его под именем «*smile.tzx*». В опциях просмотра режима «Header» тип заголовка выставите «Bytes:», а вместо имени файла поставьте 10 пробелов. Стартовые адреса не трогайте, пусть будут 0 по умолчанию.

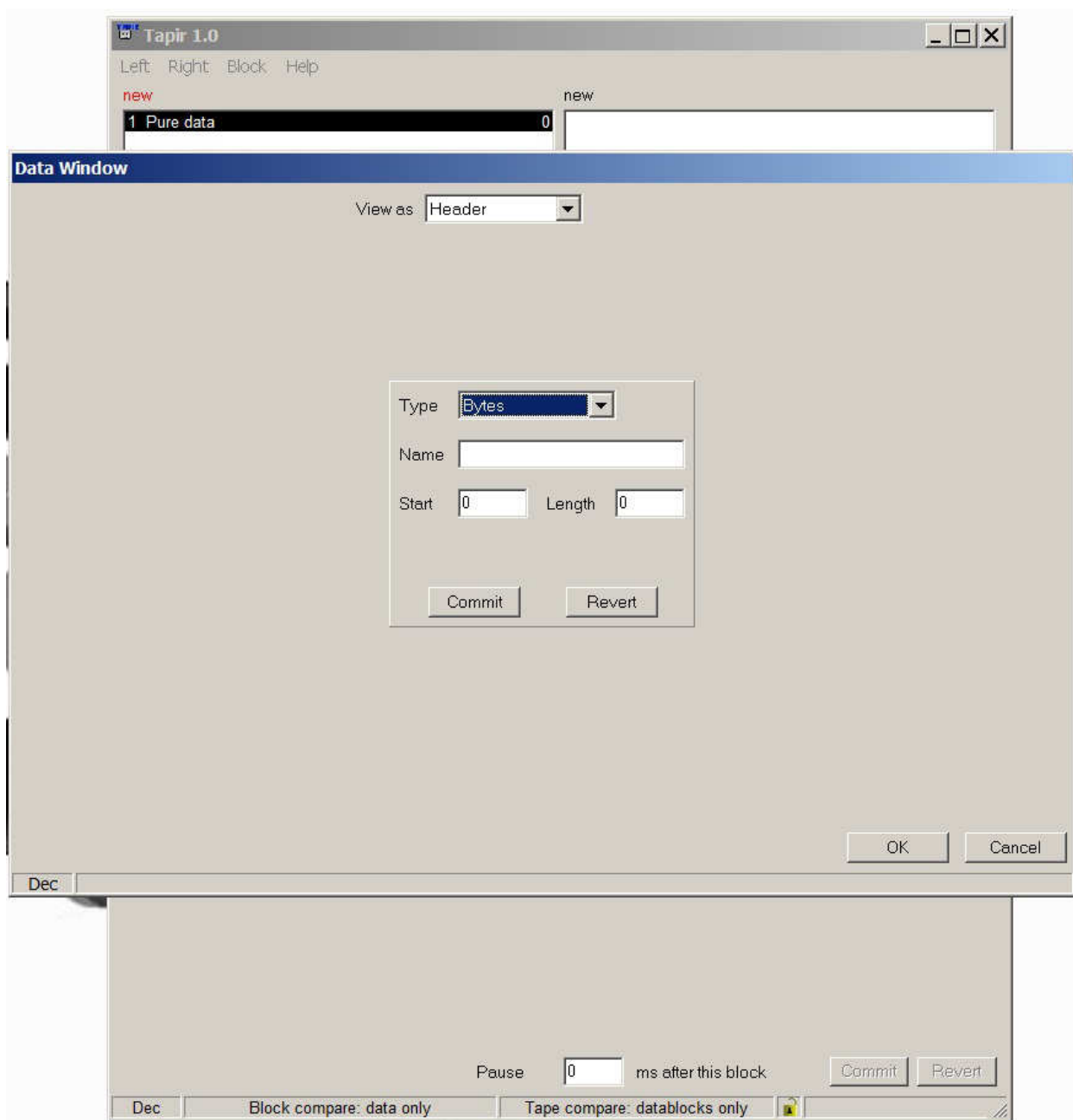


Рис. 5403. Создание шаблона заголовка «Bytes:» для вставки в его имя графических символов.

После создания блока выйдете в основную программу. Так как мы будем создавать укороченный пилот-тон, то в окне «*Block type*» выберите «*Turbo speed*». Сразу появятся окошки с редактированием параметров сигнала. В окошке «*Pilot lenght*» оставьте длину «3223», и нажмите «*Commit*» в нижнем углу. Сохраните его под именем «*sm-scr.tzx*»:

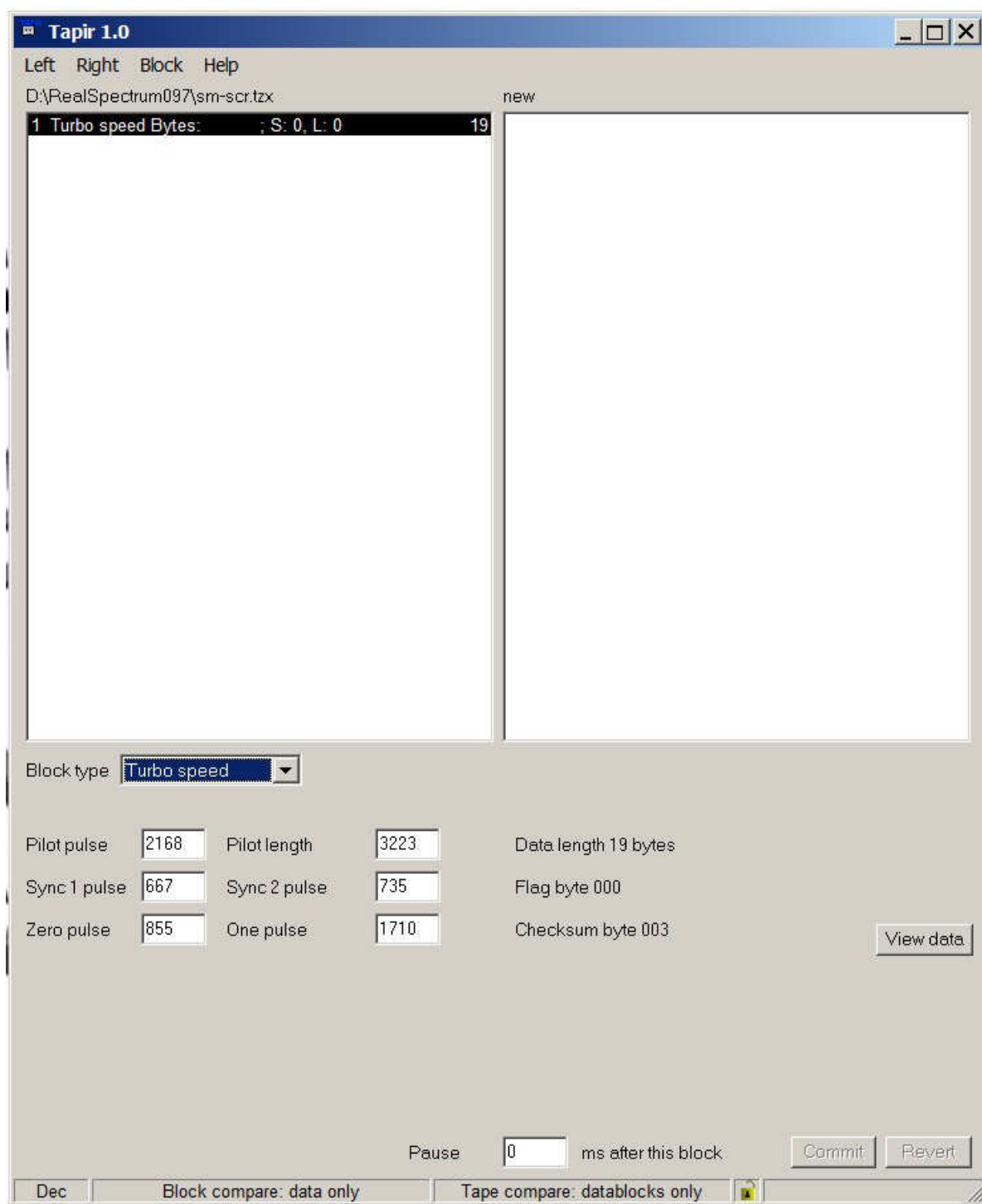


Рис. 5404. Tapir. Готовый шаблон заголовка с укороченным сигналом пилот-тона.

Теперь не закрывая программу, откройте папку с сохраненным файлом «*sm-scr.tzx*». Перетащите мышкой в правое окно программы Tapir и отпустите кнопку. Там появится копия созданного файла:

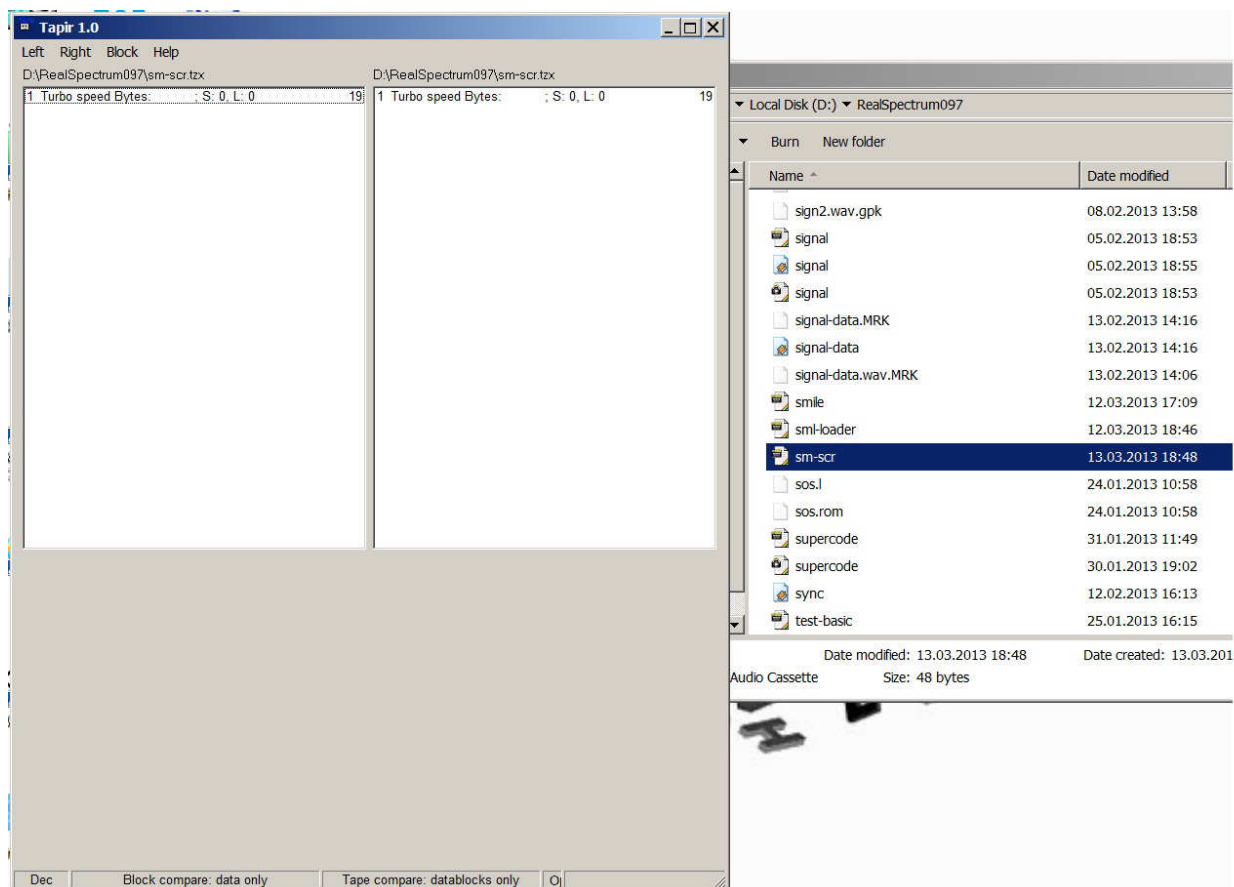


Рис. 5405. Tapir. Процесс добавления блока к существующему. Перетаскивание файла в соседнее окно.

А теперь мышкой, с зажатой левой кнопкой, перетащите заголовок из окна «Right» в «Left». Заголовок добавится в окно «Left», не затерев существующий, а добавившись ниже, строкой номер 2:

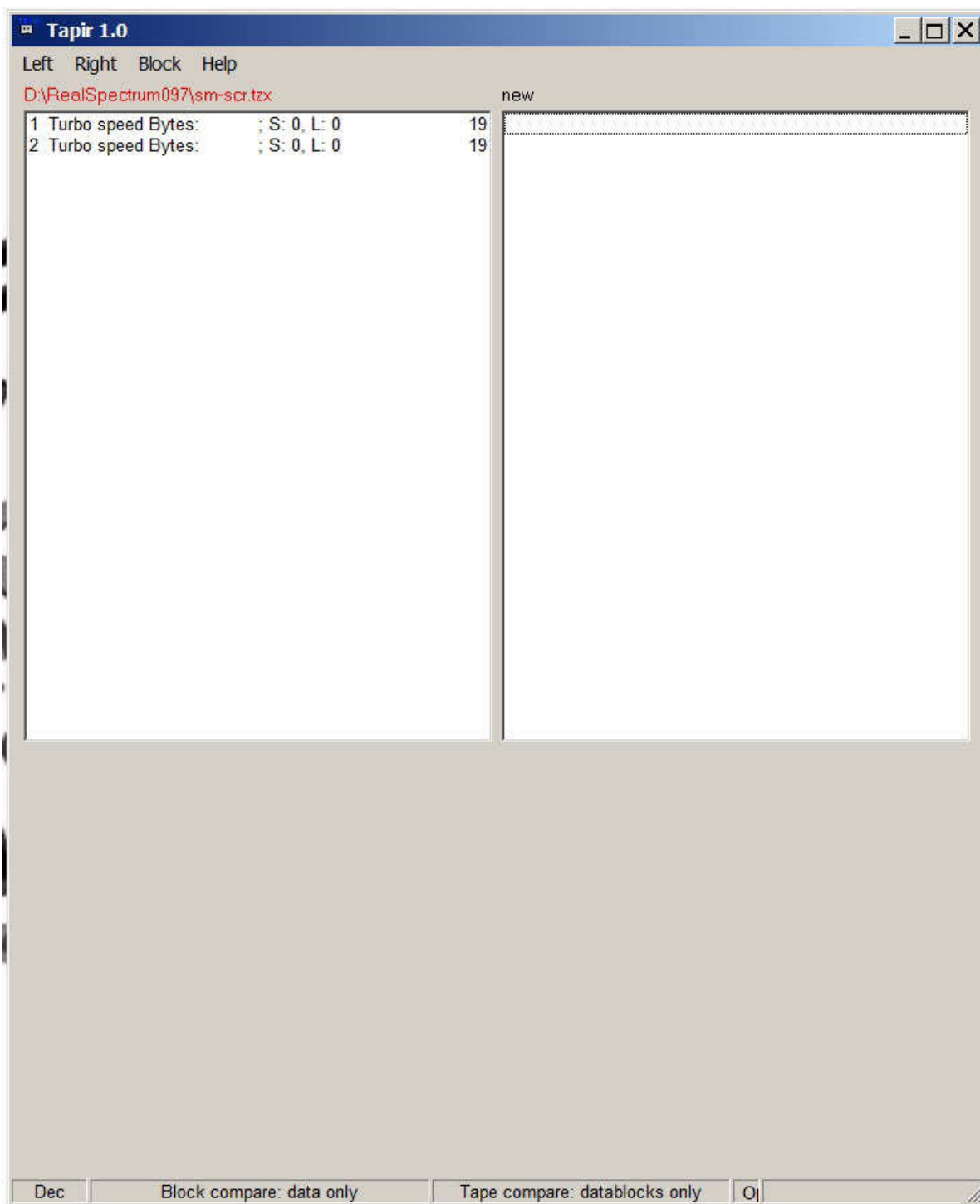


Рис. 5406. Tapir. Процесс добавления блока к существующему. Формирование многоблочного файла .tzx

Повторите манипуляции двойного перетаскивания до тех пор, пока не накопируете столбец из шести заголовков «Bytes:». Следом, таким же образом, перенесите «sml-loader.tzx», добавив файл в окно «Right», а затем, по очереди перетащив заголовок и блок данных в «Left». А в самом конце, точно также, добавьте «photo-pict.tzx» У вас получится следующее:

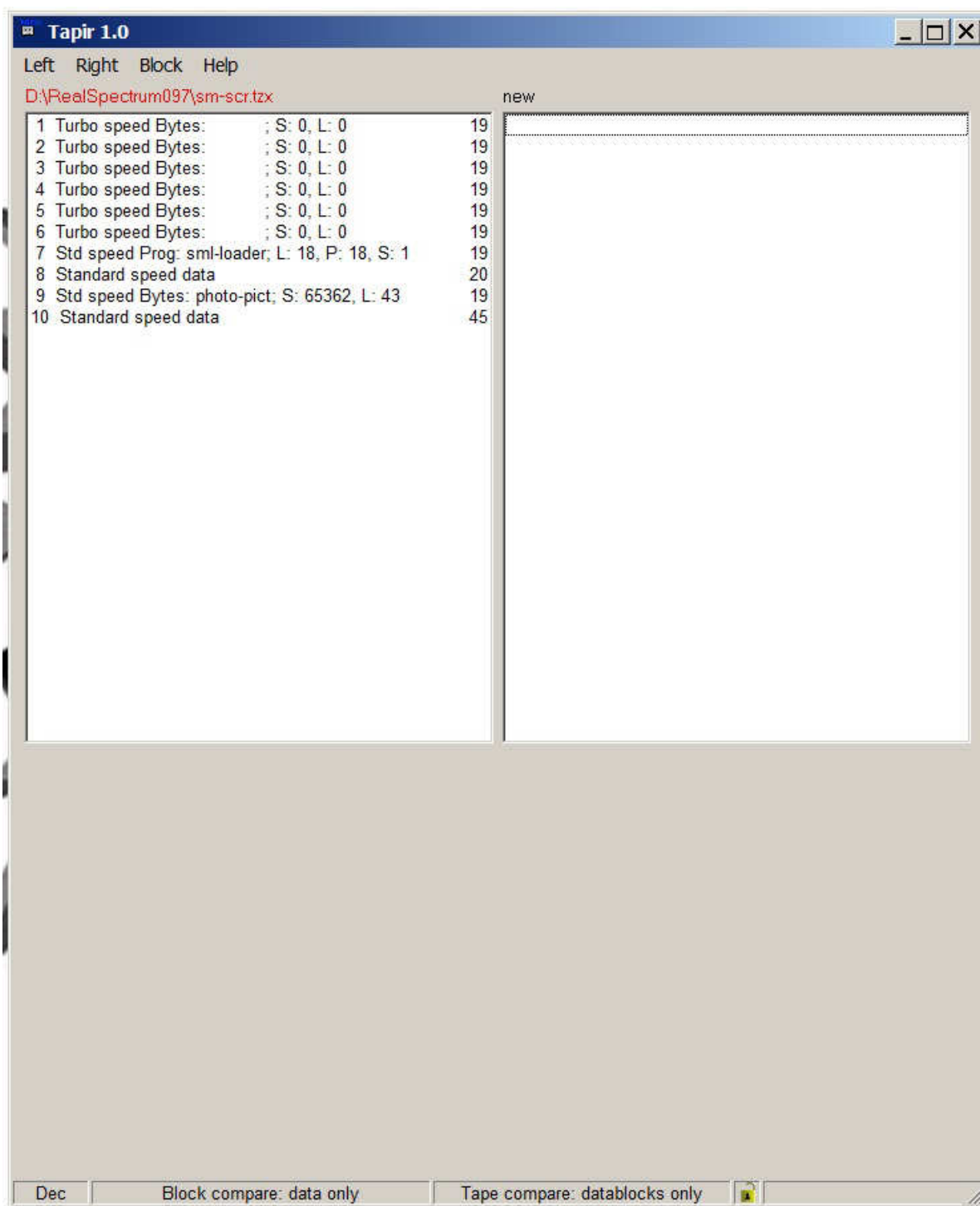


Рис. 5407. Tapir. Начерно сформированный сложный файл из заголовков и программ.

Теперь сохраним полученный сложный блок («Left» → «Save») и приступим к редактированию имен файлов заголовков, которыми и будет рисоваться картинка. Выделите и откройте первый заголовок, нажав «View data». Программа автоматически войдет в режим «Header», и предложит редактировать данные. Воздержитесь от редактирования в этом режиме, ничего не меняя, переключитесь в режим «Dump», и замените 10 пробелов (байты с кодом «032») первой строчкой рисунка смайлика, введя последовательно числа: 22, 0, 0, 137, 131, 131, 131, 131, 131, 134. Сохраните заголовок, нажав «OK». Строчка с блоком заголовка покраснела, потому что контрольная сумма поменялась, и значение проверочного байта стало неактуальным. Исправлением займемся позже. Сейчас выделите строчку 2, и откройте в режиме «Dump» следующий заголовок «Bytes:». Замените байты имени файла следующей строкой смайлика: 22, 1, 0, 138, 32, 32,

32, 32, 32, 133. Снова сохраните изменения. Исправьте имя 3-го заголовка, введя значения: 22, 2, 0, 138, 32, 130, 32, 129, 32, 133. Для 4-го заголовка введите - 22, 3, 0, 138, 32, 32, 32, 32, 133, для 5-го - 22, 4, 0, 138, 132, 32, 32, 32, 136, 133, и наконец, для 6-го - 22, 5, 0, 138, 32, 131, 131, 131, 32, 133.

Седьмую строчку рисунка вставим в заголовок «*Program:*» самого загрузчика, вместо имени «*sml-loader*». В него вставим значения: 22, 6, 0, 134, 140, 140, 140, 140, 140, 137 и сохраним изменения. В итоге у вас должно получиться следующее:

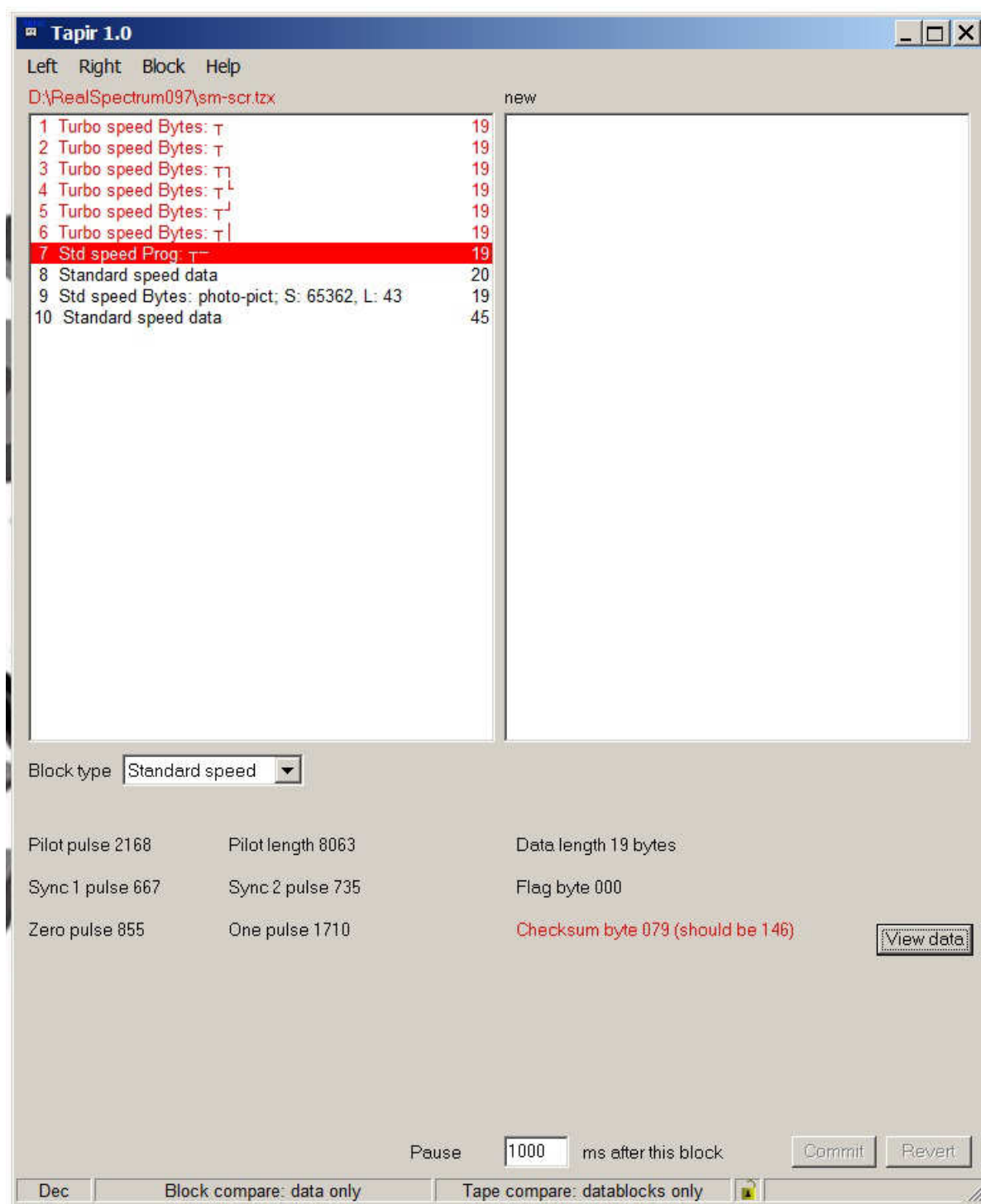


Рис. 5408. Tapir. Сформированный графический рисунок смайлика в именах заголовков.

Теперь перейдем к заключительному заголовку «*Bytes:*» с именем «*photo-pict*». Он не должен портить рисунок, поэтому его стоит сделать универсальным самозатирающимся, независимо от местоположения на экране. Например, управляющими

символами «курсор влево», отодвинем позицию заголовка на 7 клеток влево, далее, управляющим кодом **ТМК** установим белый цвет, а команда **BORDER** белым по белому точно накроет выскочивший тип заголовка «Bytes:». Для этого в строке 9, в заголовок блока, вместо «*photo-pict*» введем значения: 8, 8, 8, 8, 8, 8, 8, 16, 7, 231 и сохраним изменения. В результате строки должны выглядеть следующим образом:

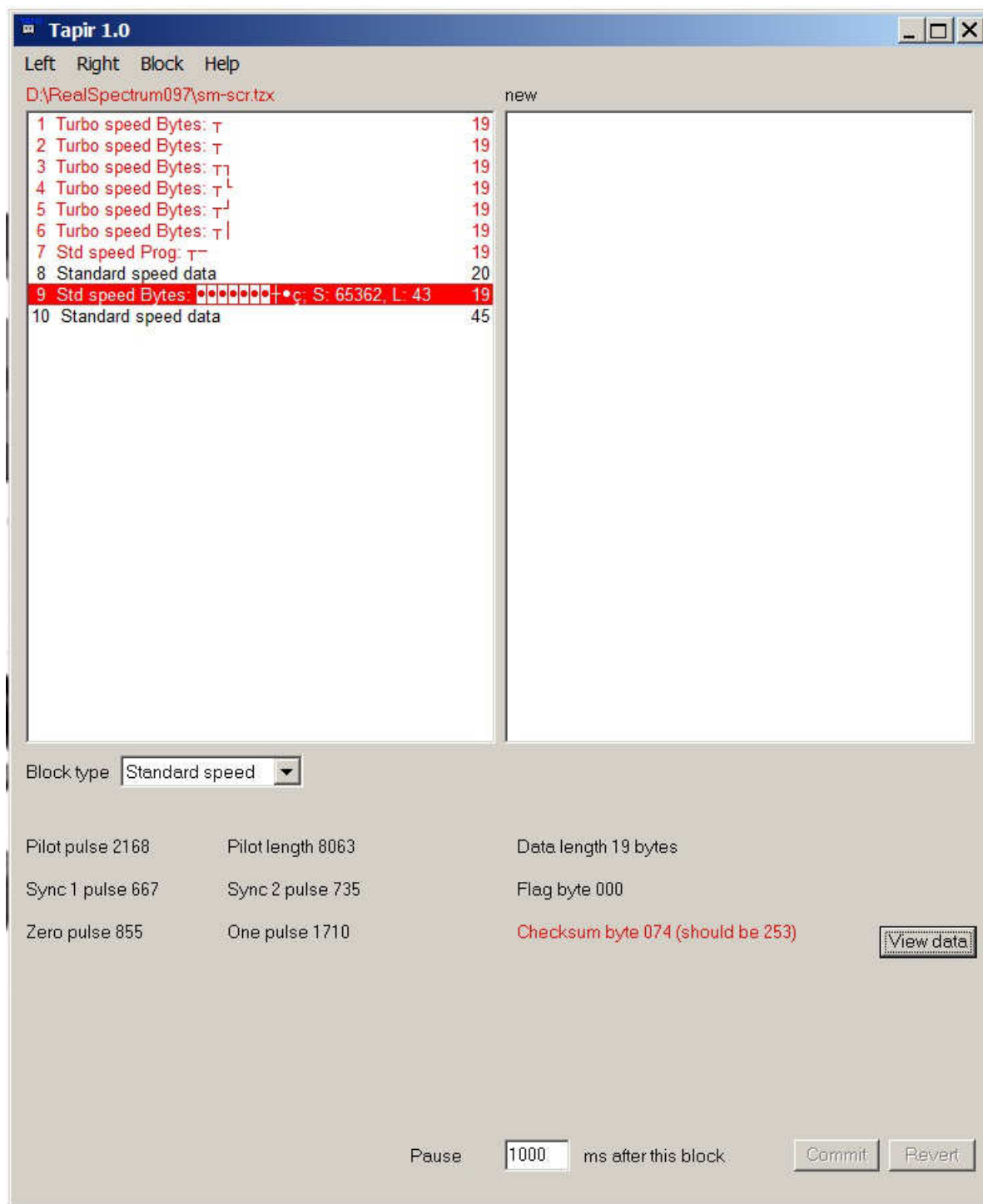


Рис. 5409. Tapir. Формирование заголовка, маскирующего самого себя.

После этого удалим все паузы в промежутках между блоками №7, №8, №9, и №10, изменим значения «1000» на «0» в окне «*Pause ... ms after this block*». Для этого выберите блок №7, установите значение «0» в окошке, и нажмите «*Commit*». Те же действия проделайте со строками №8, №9 и №10. Проверьте, чтобы для каждого блока стояло значение «0». Далее, выделим блок «*Program:*», в окошке «*Block type*», поставим «*Turbo*

speed», и изменим значение длины пилот-тона на «3223». То же самое действие сделаем с заголовком «Bytes:» в строчке 9.

Теперь начнем исправлять байты контрольной суммы во всех блоках, выделенных красным цветом. Выделите первый. В сообщении видим: «Checksum byte 003 (should be 153)». Откройте заголовок, и в режиме «Dump» введите нужное значение. Заголовок почернеет. Аналогично исправьте все остальные заголовки. После всех исправлений, когда строчки почернеют, сохраните файл «Left» → «Save». Должно получиться так:

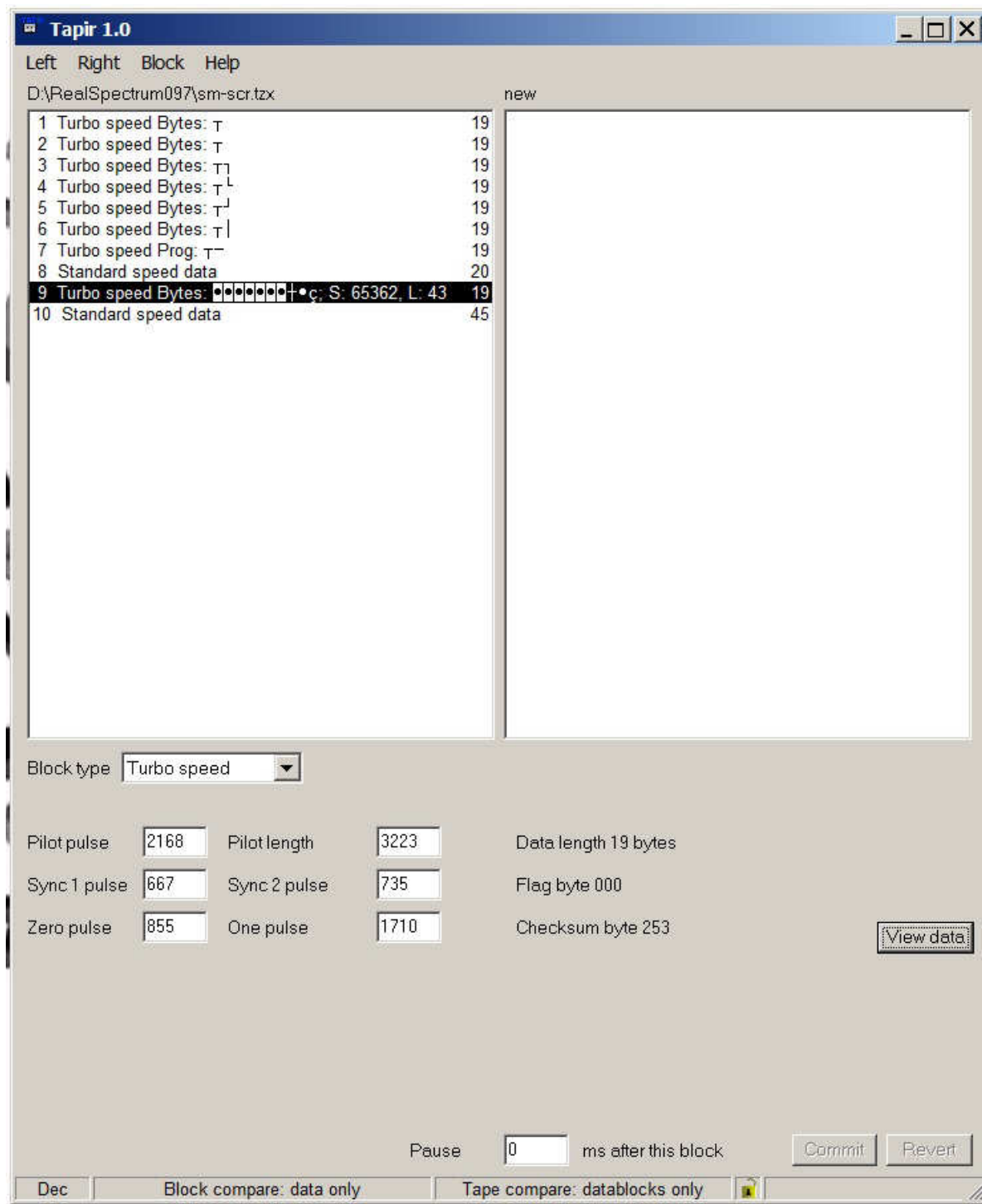


Рис. 5410. Tapir. Исправленные байты контрольной суммы. Окончательный вид готовой программы.

Закройте программу, откройте полученный файл в Spectaculator, набираем LOAD ""ENTER, и начинаем наблюдать за процессом загрузки. Длина пилот-тона будет

короткой, блоки будут проигрываться без промежутков, последовательно рисуя картинку на экране:

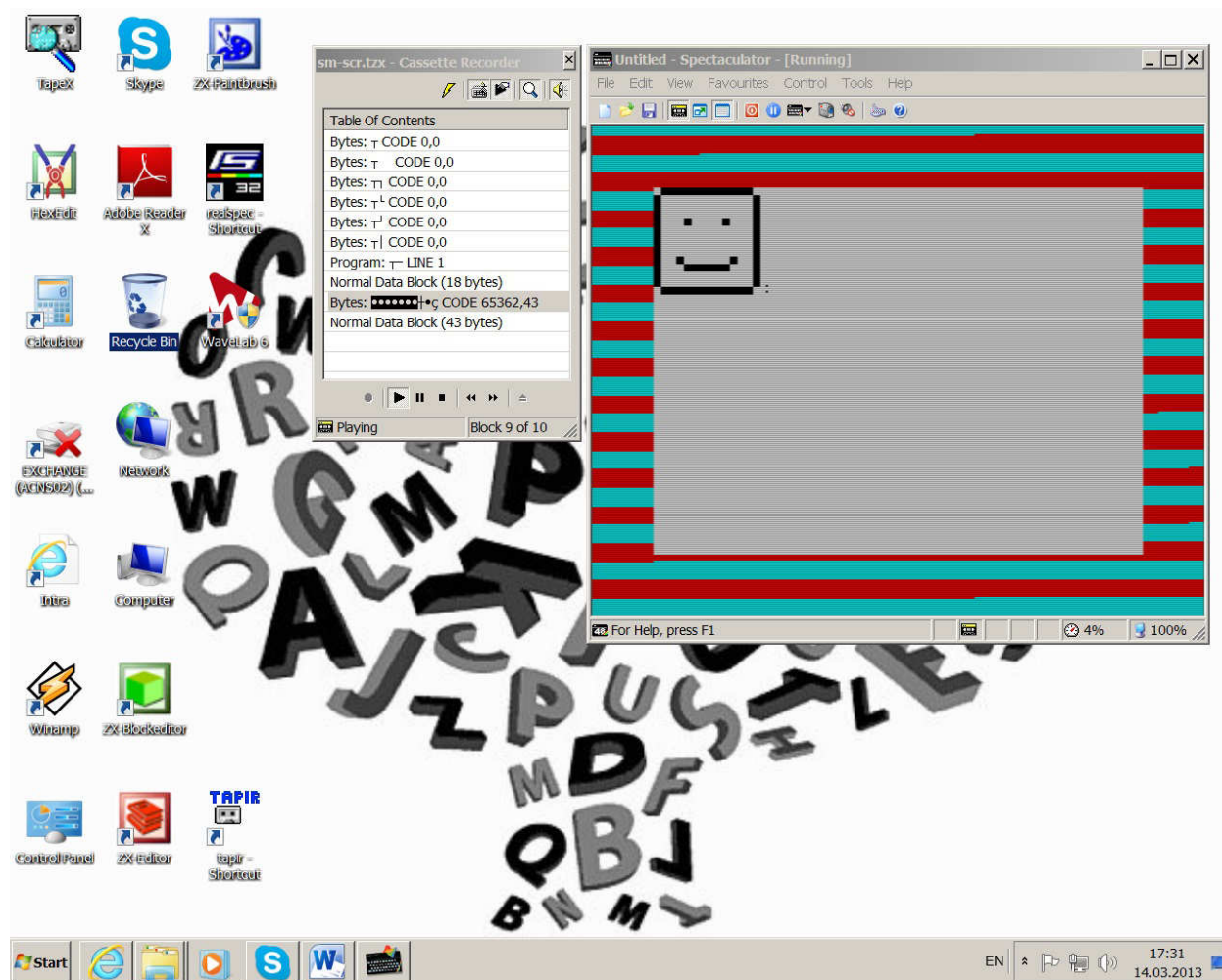


Рис. 5411. Проверка созданной программы на эмуляторе. Процесс загрузки картинки заголовками.

После загрузки программа выдаст на экран `OK, 1:2`, и выйдет в BASIC. Сотрите изображение, нажав `ENTER`. В строке программы после `REM` будет стоять напоминание, что просмотреть сфотографированную картинку можно, набрав `RANDOMIZE USR 30000`. Сделав это, на экране снова появится сохраненный смайлик, прорисованный заголовками.

ЧАСТЬ 6.

Работа с сигналом и звуком загружаемых данных.

В этой главе все внимание будет уделено загрузке, звукам и детальному изучению структуры сигнала. Большая часть экспериментов будет проводиться в звуковом редакторе.

Глава 1.

Создание мелодии из звукового сигнала загружаемых данных.

Краткое содержание: зависимость звука от значений байта, создание мелодии из бессмысленных данных.

Многие слышали, и наверняка подсознательно запомнили, как звучит загрузка картинки, программные данные, текстовая информация, машинные коды, графические данные игр, русский алфавит, или графика пользователя UDГ. Чем больше разброс значений последовательных байт данных, тем больше звук похож на шум. Мягким шумом загружаются данные, состоящие из программ в машинных кодах. Монотонный гул на одной частоте дает длинная цепочка одинаковых значений. Например «0» дают писк, «255» - гул. Таким образом, можно создать бессмысленный блок кодов из цепочек одинаковых байтов, который сигналом во время загрузки, наигрывал бы какую-нибудь мелодию. Давайте создадим такой блок данных, запишем его, и прослушаем во время загрузки. Создайте текстовый файл в блокноте windows или скопируйте по *CTRL+C* в буфер, следующую машинную программу:

ORG 30000

[illegible]

```

DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255, 255
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128
DEFB 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128

```

Теперь откройте эмулятор EmulZWin, а затем «Assembler++». Вставьте по *CTRL+V* программу, скомпилируйте ее, а в окне эмулятора, на BASIC введите подготовительную строку: `SAVE "loadmusic"CODE 30000,1240:`

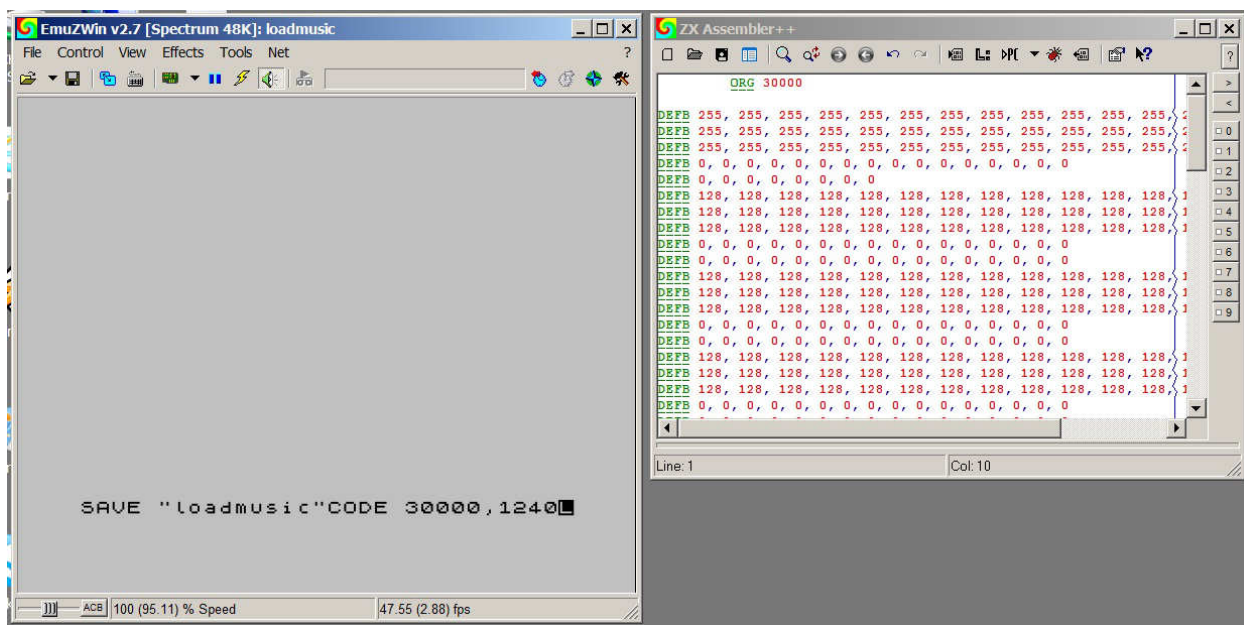


Рис. 6100. EmulZWin: Окончательный вид программы перед сохранением в .z80.

Затем сохраняем файл в «load-music.z80», и как обычно через Realspectrum создаем одноименный «load-music.tzx».

Запустите полученный файл на Spectaculator по `LOAD "" CODE` в нормальном режиме, и слушайте, как в процессе загрузки, сигналом с данными будет сыграна простенькая мелодия.

Глава 2.

Исследование стандартного сигнала загрузки в звуковом редакторе.

Краткое содержание: исследование формы и структуры сигнала, редактор Wavelab.

В предыдущей главе пробовали сделать из звука загрузки мелодию, а чуть раньше даже создавать искусственные блоки данных с помощью программы Tapir. Теперь попробуем посмотреть, что же из себя представляет программа в виде графика, и как выглядят эти таинственные «Пи-и-и-и-и Тшу-у-у-у-у». Чтобы лучше понять, наберем такую вспомогательную программу:

```

ORG 40000
DEFB 0, 0, 0, 0, 1, 1, 1, 1
DEFB 2, 2, 2, 2, 3, 3, 3, 3
DEFB 4, 4, 4, 4, 5, 5, 5, 5
DEFB 6, 6, 6, 6, 7, 7, 7, 7
DEFB 8, 8, 8, 8, 9, 9, 9, 9
DEFB 10, 10, 10, 10, 11, 11, 11, 11
DEFB 12, 12, 12, 12, 13, 13, 13, 13
DEFB 14, 14, 14, 14, 15, 15, 15, 15
DEFB 127, 127, 127, 127, 128, 128, 128, 128
DEFB 254, 254, 254, 254, 255, 255, 255, 255
DEFB 0, 1, 2, 3, 4, 5, 6, 7
DEFB 8, 9, 10, 11, 12, 13, 14, 15
DEFB 127, 128, 254, 255

```

Скомпилируем с помощью эмулятора EmulZWin, создадим из нее «signal.tzx» с помощью Realspectrum, и наконец, создадим *.wav файл с помощью программы Tapir, как

было описано в предыдущих главах. Или создадим искусственный файл из данных, как описывалось в четвертой части.

Сначала вспомним, в какой последовательности, и из каких основных «деталей» состоит загружаемый блок:

Block type: Standard speed		
Pilot pulse 2168	Pilot length 3223	Data length 102 bytes
Sync 1 pulse 667	Sync 2 pulse 735	Flag byte 255
Zero pulse 855	One pulse 1710	Checksum byte 001

Pause 1000 ms after this block

Рис. 6200. Стандартные параметры звуковых сигналов загрузки данных ZX-Spectrum.

Первым идет сигнал пилот-тона, и как видно из данных, имеет самую большую размерность (*Pilot pulse 2128*). Следом идет короткий синхроимпульс. В идеале, верхняя часть которого 667, а нижняя 735 (*Sync pulse 1 - 667* и *Sync pulse 2 - 735*). А дальше уже идут данные из цепочки чередующихся битов «0» и «1». Бит «0» имеет размерность 855 (*Zero pulse 855*), а бит «1» - 1710 (*One pulse 1710*). Ну и как убедились в прошлой главе, первым байтом перед программой, после синхроимпульса, будет стоять маркер «255» для данных, или «0» для заголовка. Замыкает программу проверочный байт контрольной суммы. В целях экономии места контрольная сумма, независимо от длины программы, представляет собой 1 байт. Начиная с маркера заголовка, каждое число накладывается друг на друга с помощью операции XOR.

Перейдем к практике. Откроем полученный файл с помощью звукового редактора, в котором есть функция ручного редактирования сигнала (карандаш, *pen tool* или что-то подобное), и посмотрим на практике, что собой представляет сигнал. Для этих целей мне показался удобным звуковой редактор Wavelab. Установите и настройте редактор.

Откроем в нем созданный «*signal.wav*» файл, и увеличив фрагмент, посмотрим на структуру сигнала пилот-тона:

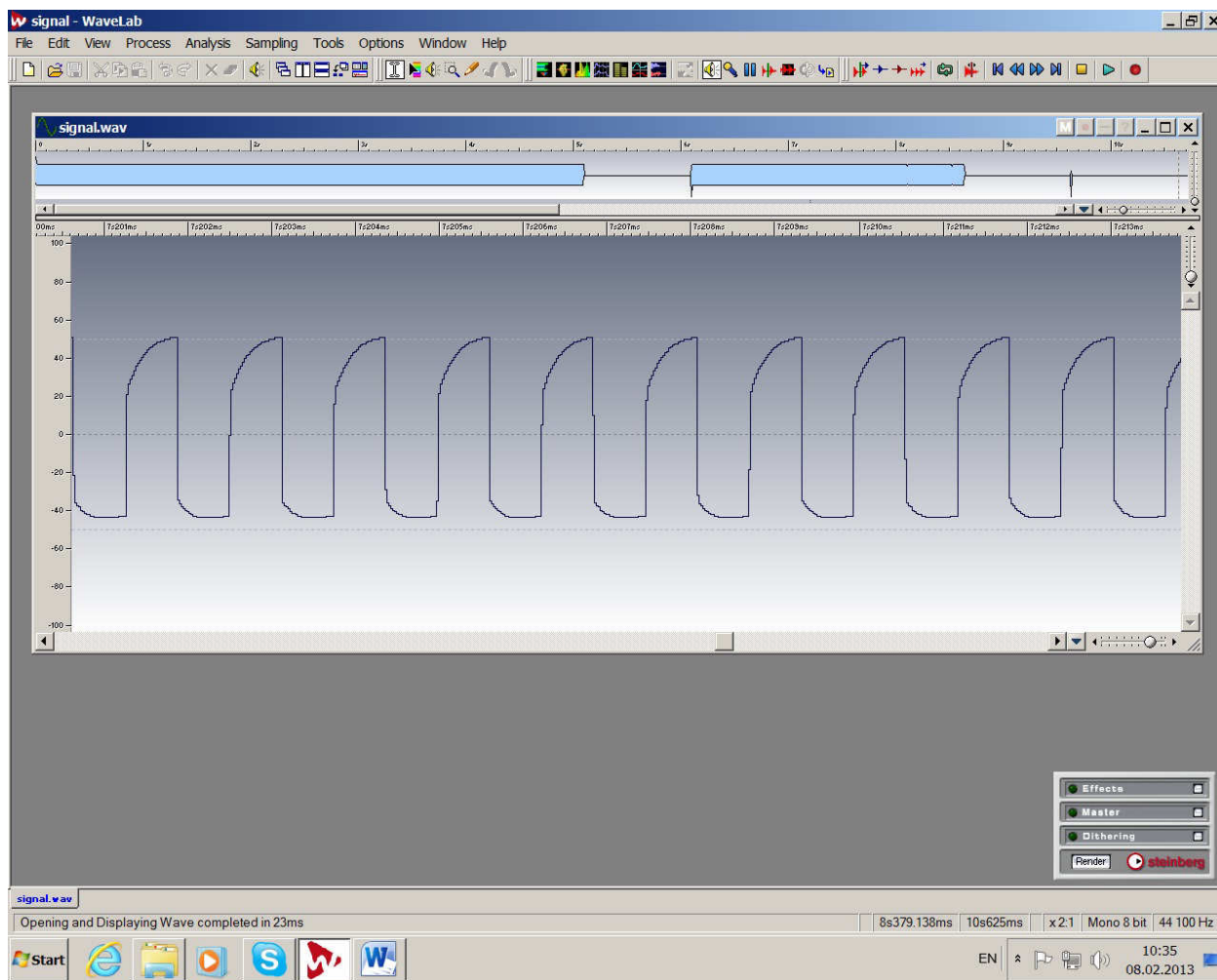


Рис. 6201. Редактор Wavelab. Графическое представление сигнала пилот-тона.

Сигнал пилот-тона представляет из себя столбики. У всех скруглены левые верхние, и правые нижние края, как будто по ним прошлись напильником. Как видно на графике, даже искусственный сигнал, созданный программой неидеален. Хотя по идее, каждый столбик должен быть одинаковый. Теперь нетрудно представить какой искаженный сигнал считывал настоящий Спектр с потертой и зажеванной отечественной магнитофонной кассеты. И самое удивительное, воспринимал же. А теперь посмотрим на сигнал данных заголовка, на стыке пилот-тона, синхроимпульса, маркера и программных данных. Выглядеть это место, на примере заголовка «Bytes:» будет вот так:

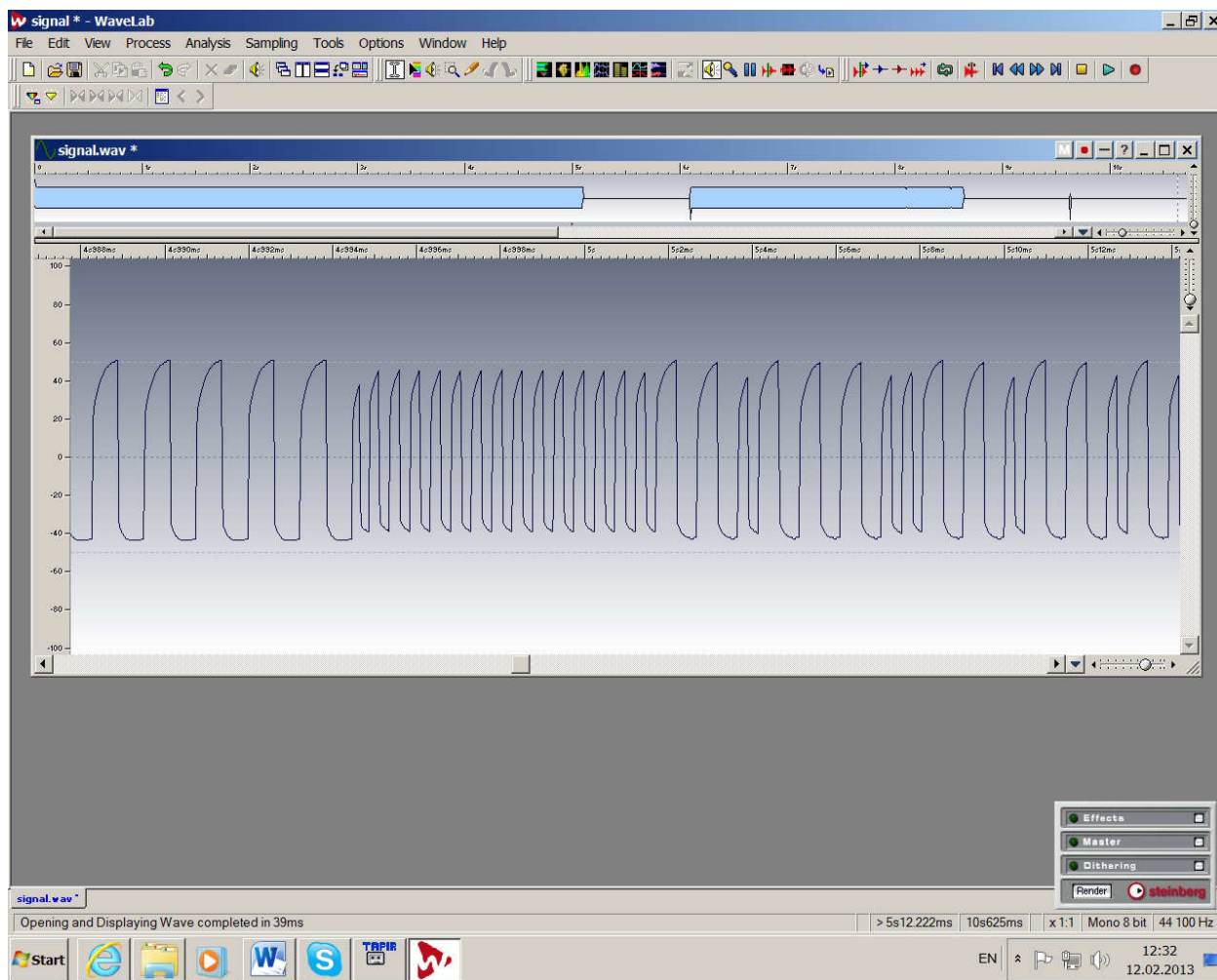


Рис. 6202. Фрагмент сигнала заголовка «Bytes:» на стыке пилот-тона, синхрои́мпульса и битов данных.

Слева видим 5 широких столбиков пилот-тона, следом за ними прицеплен столбик синхрои́мпульса, самый низкий по высоте. Далее идет чередование всего 2-х видов столбиков. Все они также со сточенными левыми краями. Сравнив с данными программы, становится ясно, что компьютер считывает данные побитно. Широкий и высокий столбик – это бит «1», узкий и пониже – бит «0». Таким образом, байт записывается 8-ю столбиками. Следовательно, после бита синхрои́мпульса, идут 8 битов «0». Это маркер, который в данном случае указывает, на заголовок. Следующий байт имеет значение 3, означая тип заголовка «Bytes:» и далее его спектрмское имя. Таким образом, после синхрои́мпульса мы видим чередование битов «0» и «1», которые, сцепляясь друг с другом, формируют цепочку последовательно загружаемых данных. Тогда блок данных, в том-же месте будет выглядеть вот так:

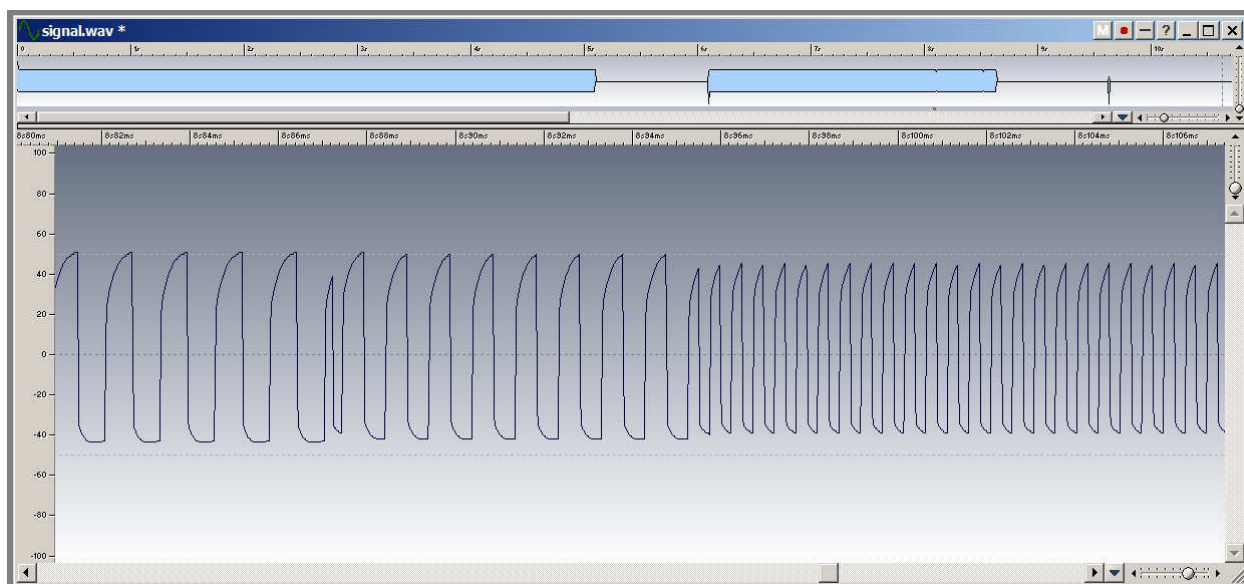


Рис. 6203. Фрагмент сигнала блока данных на стыке пилот-тона, синхроимпульса и битов данных.

Все тоже самое. После пилот-тона стоит синхроимпульс, затем 8 битов «1», означающих маркер данных «255», а далее сами данные. На данном примере это биты «0».

Глава 3. Создание блока данных в звуковом редакторе.

Краткое содержание: работа в редакторе Wavelab, создание «деталей» сигнала из неделимых звуковых частиц, пилот-тон, синхроимпульс, сигналы битов «0» и «1», сборка байтов из битов, сборка фрагментов сигнала, ручное вычисление контрольной суммы блока данных.

Изучив структуру звукового сигнала, может возникнуть закономерный вопрос: А можно ли самому вручную нарисовать сигнал из сэмплов, собрать блок, и таким образом создать собственную программу из неделимых частиц звука? Давайте попробуем провести эксперимент и создать простейший 5-ти байтный блок кодов в звуковом редакторе. Для этого сначала в редакторе снимем размеры всех необходимых элементов, а потом создадим конструктор из 4 «деталей» сигнала, формирующий стандартную загрузку.

Откроем файл «*signal.wav*» в Wavelab, увеличим и посмотрим, из сколько неделимых частиц (сэмплов) состоит сигнал, и какая его амплитуда:

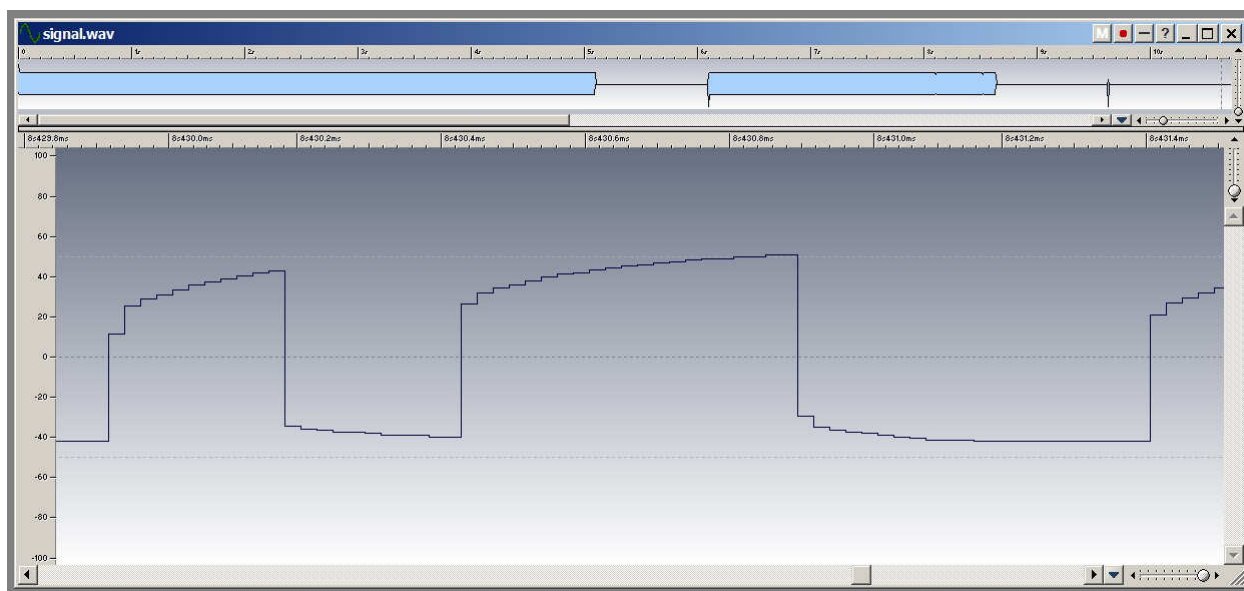


Рис. 6300. Wavelab: Увеличенные графики сигналов бита «0» (слева) и бита «1» (справа).

Слева виден сигнал, соответствующий биту «0». Он самый короткий, и состоит из 22 семплов (11 верх+11 низ) амплитуда примерно от -39,06 до +45,31. Справа виден сигнал бита 1, он состоит из 42 семплов (21 верх +21 низ) амплитуда колебаний от -42,19 до +50.

Теперь рассмотрим сигнал синхроимпульса. В таком качестве как 16 bit/44100 Hz пропорции 667 к 735 будут незначительными. Верхняя и нижняя часть этого сигнала будет практически одинаковой. Верхнюю условно возьмем 8 сэмплов, а уровень +39,07. Нижняя часть будет 9 сэмплов с уровнем -39,06. Синхронизирующий бит рядом с битом «0» выглядит так:

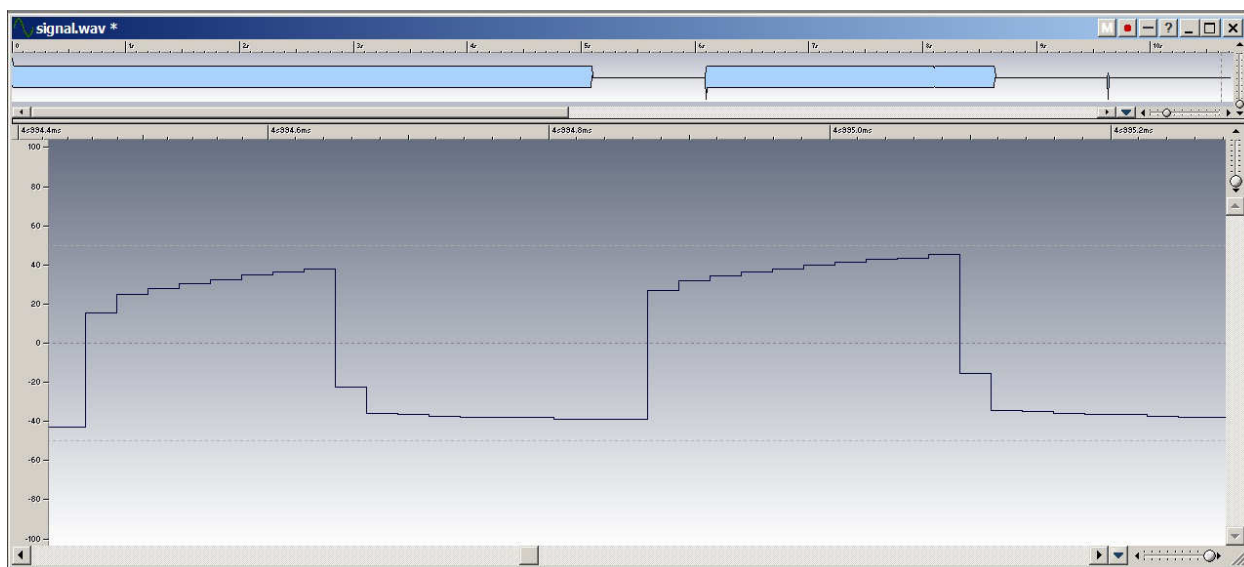


Рис. 6301. Редактор Wavelab. Синхроимпульс и бит «0» рядом друг с другом.

Синхроимпульс чуть ниже нуля, и может показаться просто бракованным битом нуля. По идее этот «бит» должен быть определенных пропорций нижней и верхней части, но учитывая качество записи/воспроизведения с магнитной ленты, возможно, такие мелочи можно опустить, и условно обе части верх и низ сделать одинаковыми. Просто во избежание недоразумений при ручной сборке сигнала за пилот-тоном следует вставить дополнительный бит, чтобы не перекосило программу. Но чтобы быть точными, все-таки

сделаем этот несимметричный бит синхронизации, наиболее реальным, согласно спецификации.

Ну и «снимем размеры» сигнала пилот-тона. Он самый широкий, состоит 54 сэмплов (27 верх+27 низ) амплитуда от +50,78 до -43,75.

Теперь, когда размеры сняты, чтобы создать элемент, манипулируя только звуком, нам нужно нарисовать по одному неделимому сэмплу верха и низа, состыковывая которые получим полноценный элемент идеальной прямоугольной формы. Соберем сигнал именно прямоугольной формы, и не будем скруглять кончики.

Создадим неделимые фрагменты пилот-тона, синхроимпульса, бита «0» и бита «1». Для этого создадим 4 новых звуковых файла 44100 Hz, 16 бит моно.

Наполним файлы звуковыми частицами, скопировав в каждый файл по два произвольных сэмпла. Карандашиком (*pencil*) поднимем уровень верхней и нижней частей до нужных значений. Получились файлы, размером в 2 сэмпла, с эталонными значениями уровней громкости будущих сигналов:

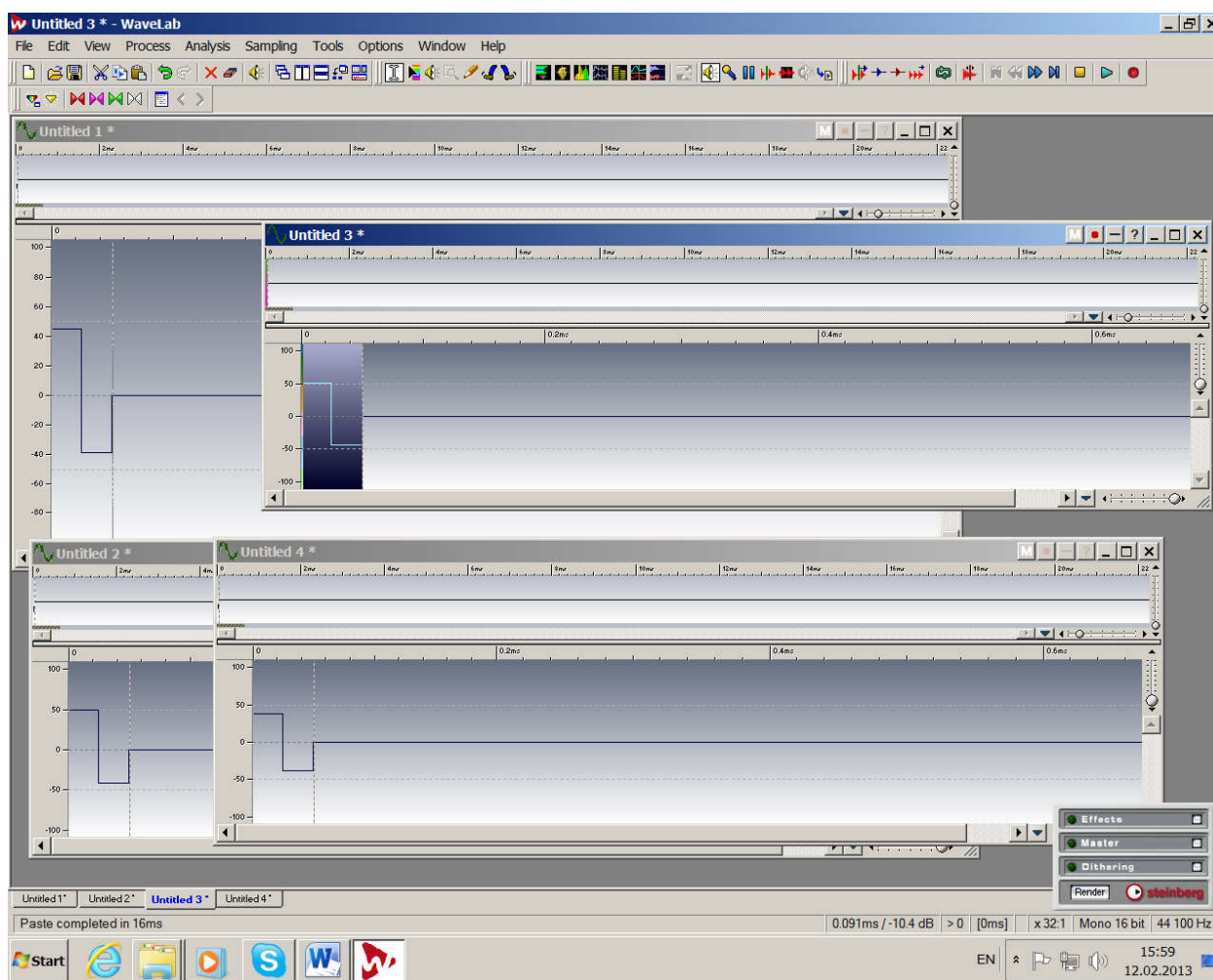


Рис. 6302. Редактор WaveLab. Заготовки фрагментов сигналов с эталонными значениями верха и низа.

Теперь в каждом таком шаблоне размножим верхний и нижний сэмпл, с помощью *CTRL+C* и *CTRL+V* нужное количество раз. Например, для сигнала синхроимпульса нужно 8 верхних частей, и 9 нижних, для бита «0» - 11 верхних, и 11 нижних.

Полученные шаблоны заготовок прямоугольных сигналов, сохраним под именами «*bit0.wav*», «*bit1.wav*», «*pilot.wav*» и «*sync.wav*»:

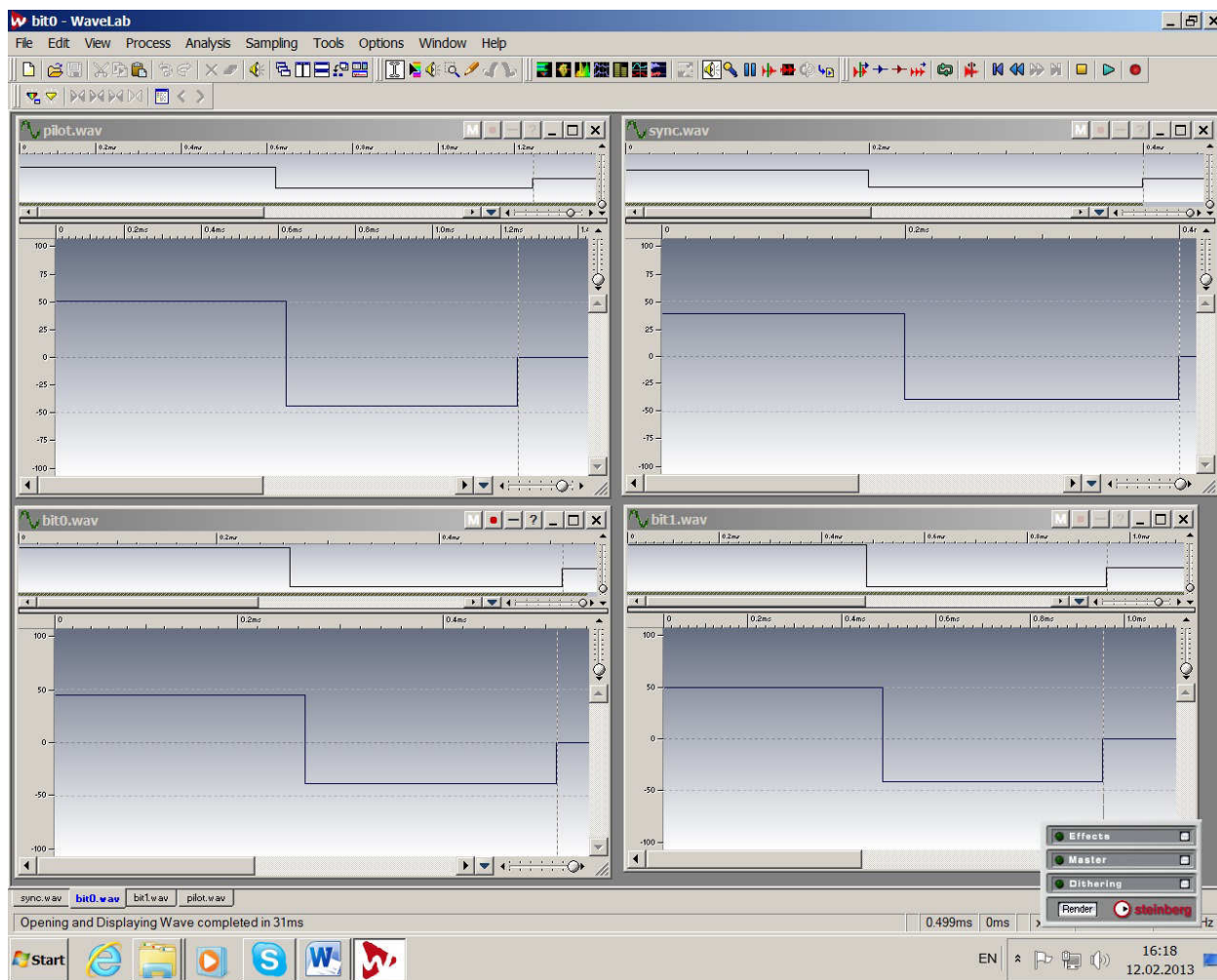


Рис. 6303. Редактор Wavelab. Готовые прямоугольные шаблоны сигналов пилот, синхронимпульс, «1» и «0».

Необходимый набор «деталей» для построения сигналов с информацией, готовы. Теперь из них можно строить цепочку битов, копируя из буфера обмена нужный «кубик».

Сначала создадим пилот-тон длиной 2 секунды. Возьмем заготовку «*pilot.wav*», создадим новый файл, и многократно раскопируем его в цепочку, расставив фрагменты сигнала, друг за другом, чтобы получился однородный сигнал в 2 секунды. Сохраним его например: «*pilot2s.wav*»

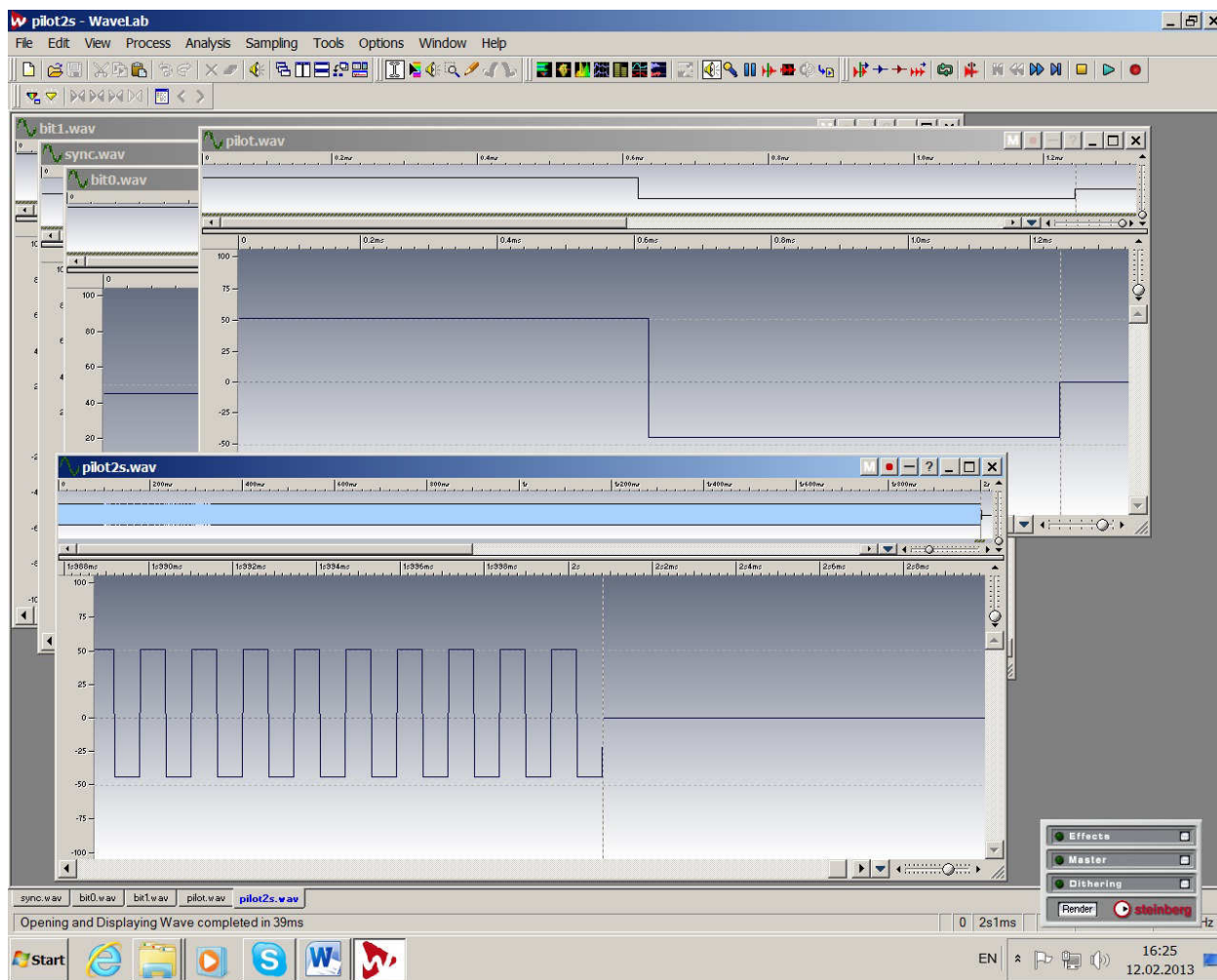


Рис. 6304. Редактор Wavelab. Готовый прямоугольный сигнал пилот-тона длиной 2 секунды.

Теперь «прицепим» в конец получившегося файла кубик бита синхроимпульса, а следом за ним 8 битов «1», формирующих байт-маркер данных «255», и вновь сохранимся. Это будет наша универсальная заготовка для любого блока данных (пилот-тон+синхроимпульс+маркер данных):

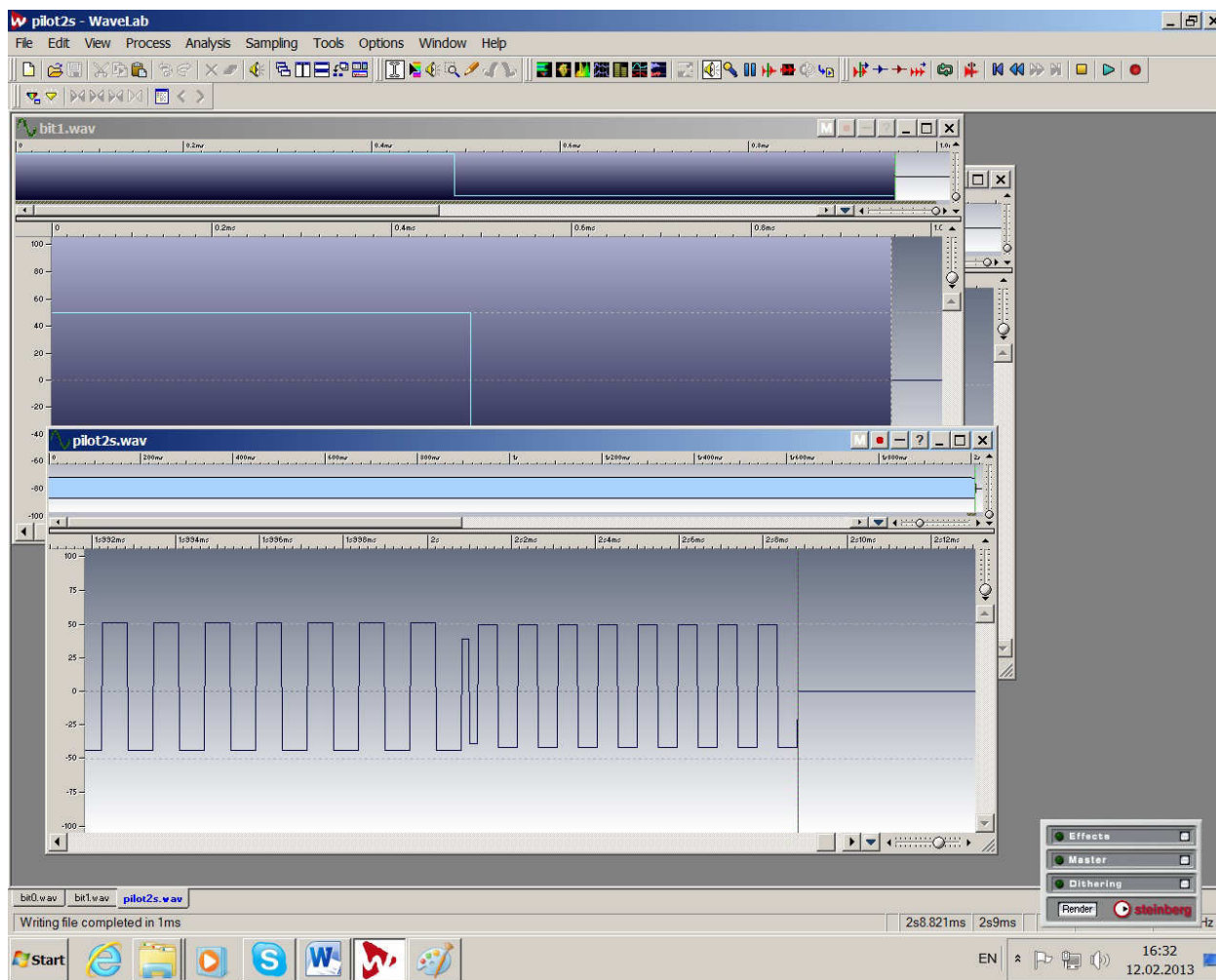


Рис. 6305. Универсальная заготовка для блока данных пилот+ синхронимпульс+маркер «255».

Теперь составим цепочку битов, имитирующих простейшую программу в машинных кодах, которая при запуске по **RANDOMIZE USR** выведет на экран фиолетовую рамку, показывая, что она верно считалась и запустилась.

Например:

```
LD A, 3
OUT (254), A
RET
```

Эта программа займет в памяти 5 байт, и будет состоять из последовательности чисел 62, 3, 211, 254, 201. Переведем их в двоичный код с помощью калькулятора windows. Спереди получившихся числовых значений добавляем недостающие нули, дополняя до 8 бит. Получилась программа, длиной в 40 бит:

00111110, 00000011, 11010011, 11111110, 11001001.

Теперь закроем все лишние файлы в редакторе, не забыв сохраниться. Оставим заготовки сигналов одиночных битов «*bit0.wav*» и «*bit1.wav*». Для формирования цепочки битов с данными создайте еще один новый файл. С помощью манипуляций с буфером обмена **CTRL+C** и **CTRL+V**, поочередно перетаскивая сигналы в новый файл, составьте цепочку из битов нужной последовательности, расставляя их друг за другом без промежутков. Сохраним его под именем «*signal-data.wav*»:

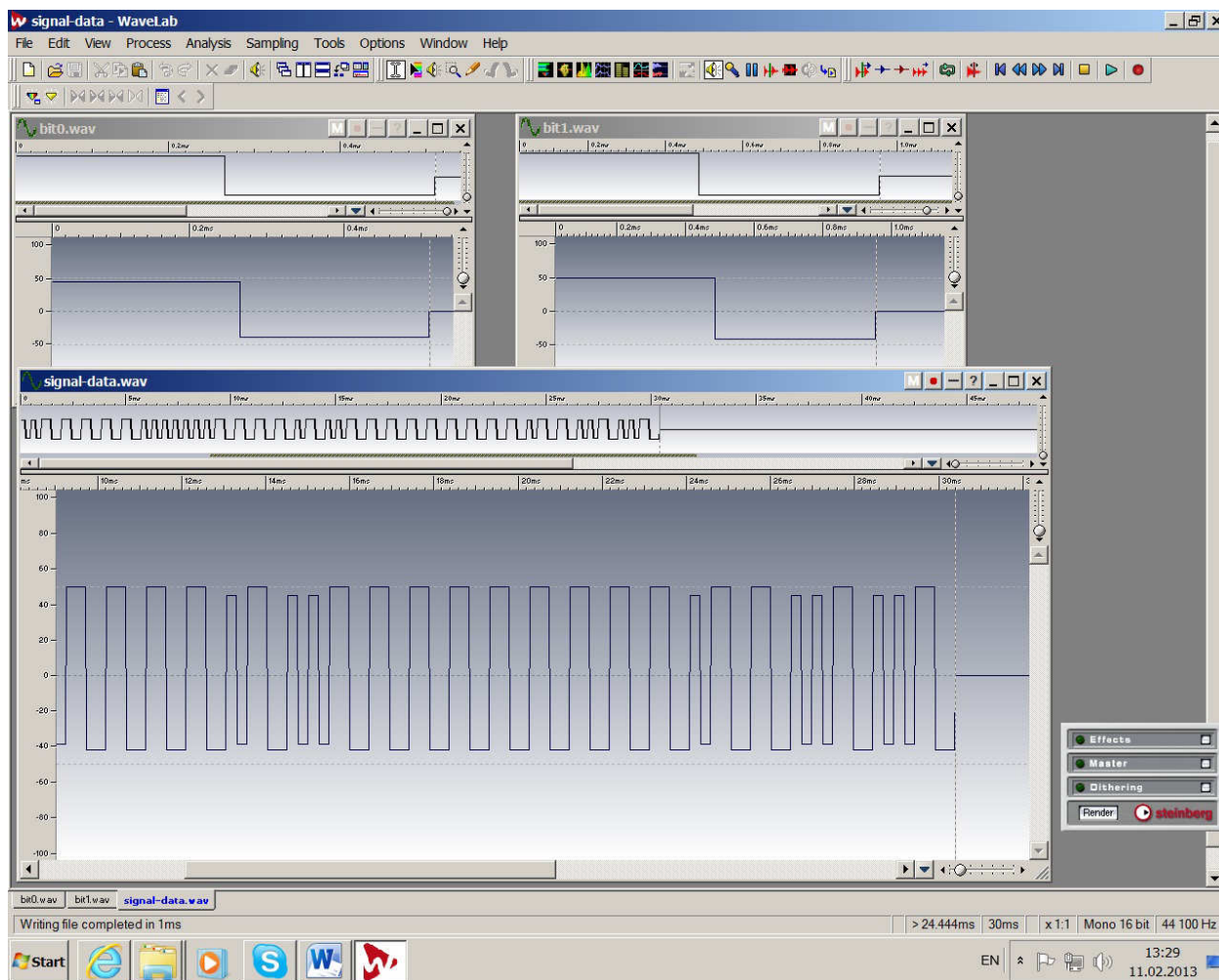


Рис. 6306. Редактор Wavelab. Побитная сборка 5-ти байтной программы из сигналов «0» и «1».

Теперь осталось подсчитать контрольную сумму блока данных, и переведа в двоичный формат, «прицепить» последним байтом, позади данных. Контрольная сумма в файлах спектрума вычисляется путем наложения на первый байт маркера, следующих, с помощью логической операции XOR. Поэтому, чтобы вычислить контрольную сумму, нужно сделать следующее действие с байтами нашей программы:

$$255 \text{ XOR } 62 \text{ XOR } 3 \text{ XOR } 211 \text{ XOR } 254 \text{ XOR } 201 = 38$$

Включив калькулятор windows 7 в режиме «Программист» (ALT+F3) последовательно пересчитаем все байты, пропустив через XOR, и в конечном итоге получим 38. Переведем его в двоичный формат, спереди дополним нулями, и получим 00100110.

Итак, вот и вычислен главный и недостающий байт. В случае его отсутствия, или ошибочного значения, после считывания данных, вместо старта программы, обычно высказывает сообщение:

R Tape loading error

В конце программы дособираем из битов байт контрольной суммы 00100110. В итоге, конец сигнала будет выглядеть вот так:

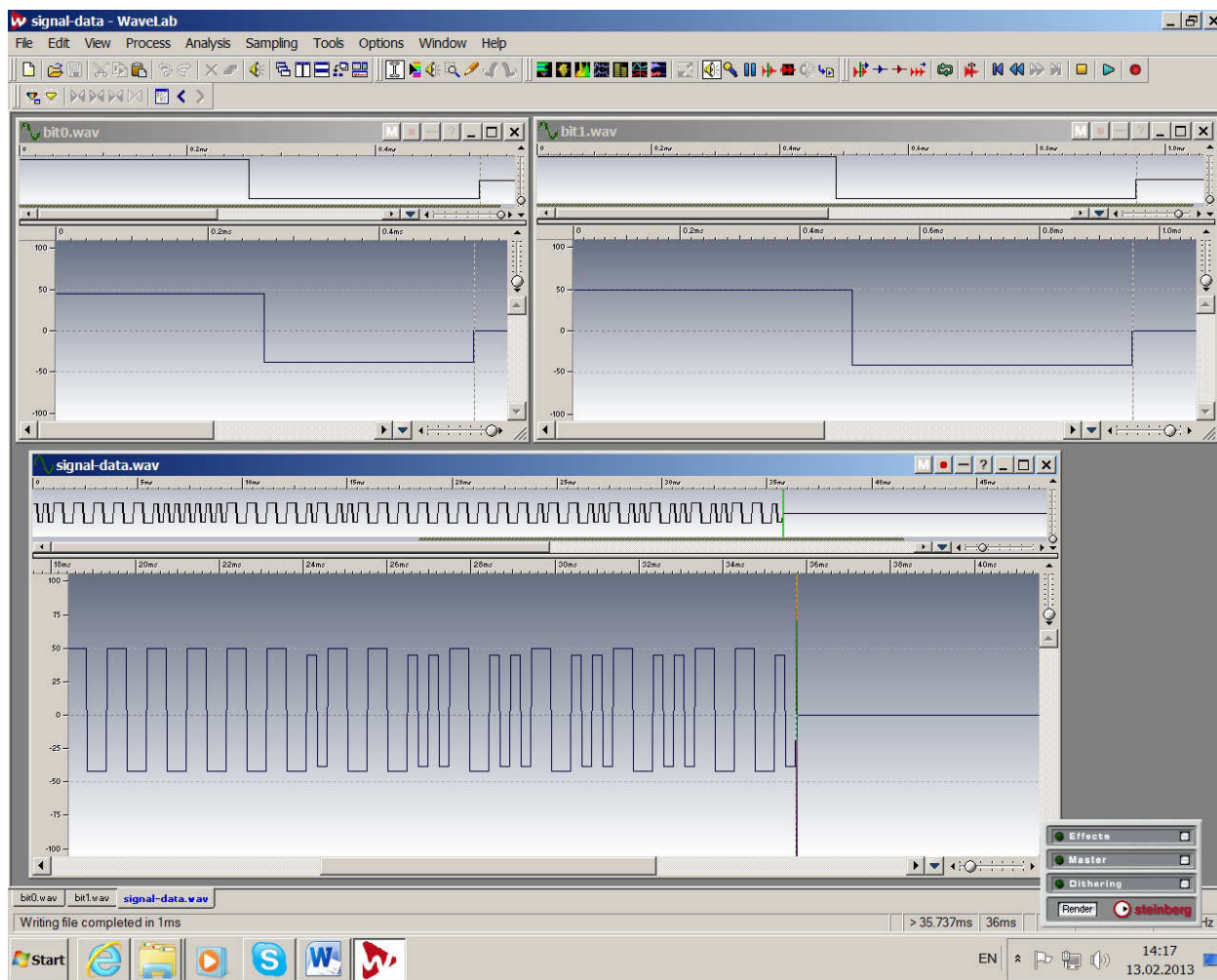


Рис. 6307. Редактор Wavelab. Конец сигнала данных с байтом контрольной суммы.

Теперь закроем «bit0.wav» и «bit1.wav», и откроем «pilot2s.wav». Скопируем созданный файл данных «signal-data», и присоединим его к концу бита синхриимпульса. Сохраним полученный файл «wavblock.wav»:

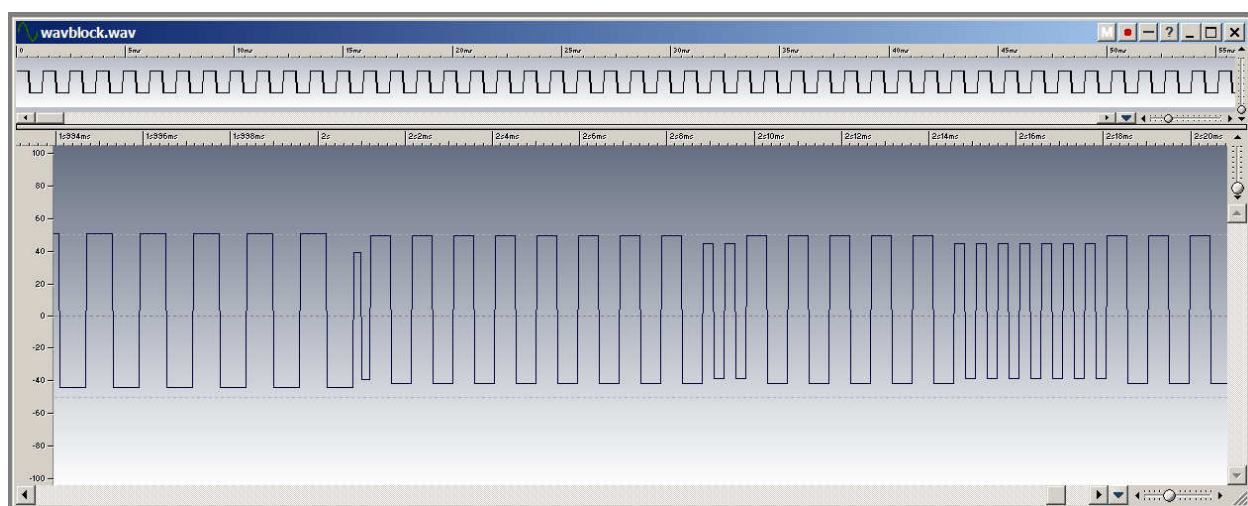


Рис. 6308. Полностью собранный блок данных с пилот-тоном, синхриимпульсом и контрольной суммой.

Вот так можно создать простенький блок данных, с помощью звукового редактора. Чисто теоретически таким образом можно создавать и длинные файлы. На практике такой метод очень трудоемкий, и поэтому полностью непригоден. Программу в 10 килобайт, а

это 80 000 бит создавать таким образом просто бессмысленно. А после этого придется вручную подсчитывать контрольную сумму на калькуляторе.

Теперь нужно проверить работоспособность полученного звукового файла. И снова обратимся к Spectaculator. Виртуальный магнитофон, помимо *.tap и *.tzh файлов прекрасно считывает и обычные *.wav файлы. Только звук должен быть 8-ми битный. Переконвертируем созданный файл из 16 в 8 бит, и сохраним его под именем «wavblock8bit.wav» («File» → «Save as...» → «Bit resolution 8 bit»)

Для воспроизведения файла, нужно просто его перетащить в виртуальный магнитофон. Но перед тем, как его проверять, этой программе требуется заголовок. Давайте, для разнообразия, создавать ничего не будем, а воспользуемся готовыми «изделиями», созданными в предыдущих главах. Заодно попробуем освоить подмену данных и заголовков, работая с виртуальным магнитофоном Spectaculator.

Парой глав раньше, с помощью программы Tapir, мы синтезировали искусственный заголовок для блока «Bytes:», рассчитанный на блок данных в 5 байт, и стартовым адресом 30000. Звуковая дорожка, которую создали в этой главе также 5 байт. Поэтому заголовок вполне подойдет для созданного файла. Принцип заключается в следующем. Сначала в магнитофон заряжаем заголовок в формате *.tzh, и ждем пока он загрузится, затем добавляем блок данных *.wav.

Откроем в Spectaculator одиночный заголовок «bl-loader.tzh» и загрузим его. После того, как выскочит заголовок, включится голубая рамка. Компьютер остановится и будет ожидать загрузки кодового блока, которого пока нет:

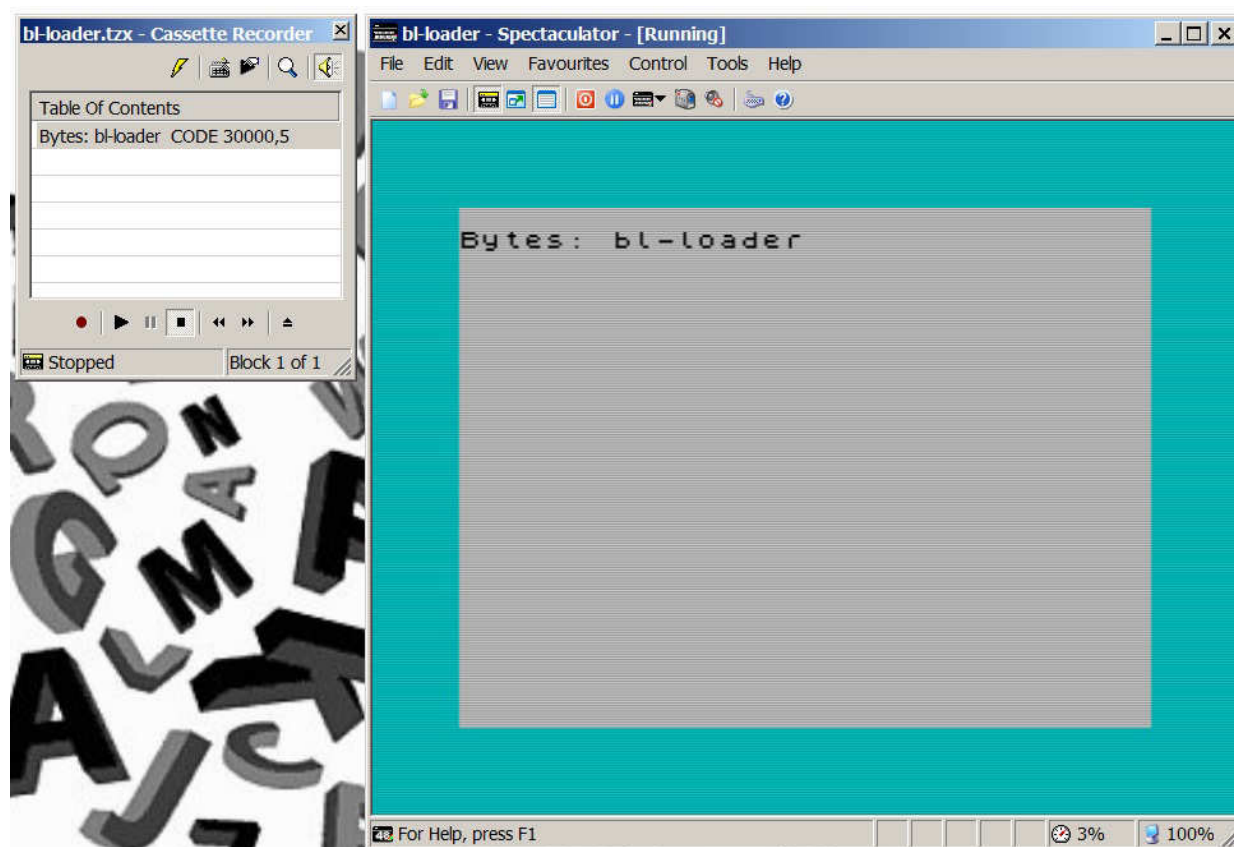


Рис. 6309. Spectaculator. Ожидание загрузки блока данных после считывания заголовка.

Теперь все просто. Перетяните в магнитофон созданный звуковой файл «wavblock8bit.wav». В магнитофоне сразу появятся сразу два блока без имени (Block 001 и Block 002), и эмулятор сразу приступит к загрузке созданного нами звуковых данных. Файлы формата *.wav эмулятор просто делит, на куски, которые последовательно считываются. Никаких заголовков и информации в магнитофоне не пишется:

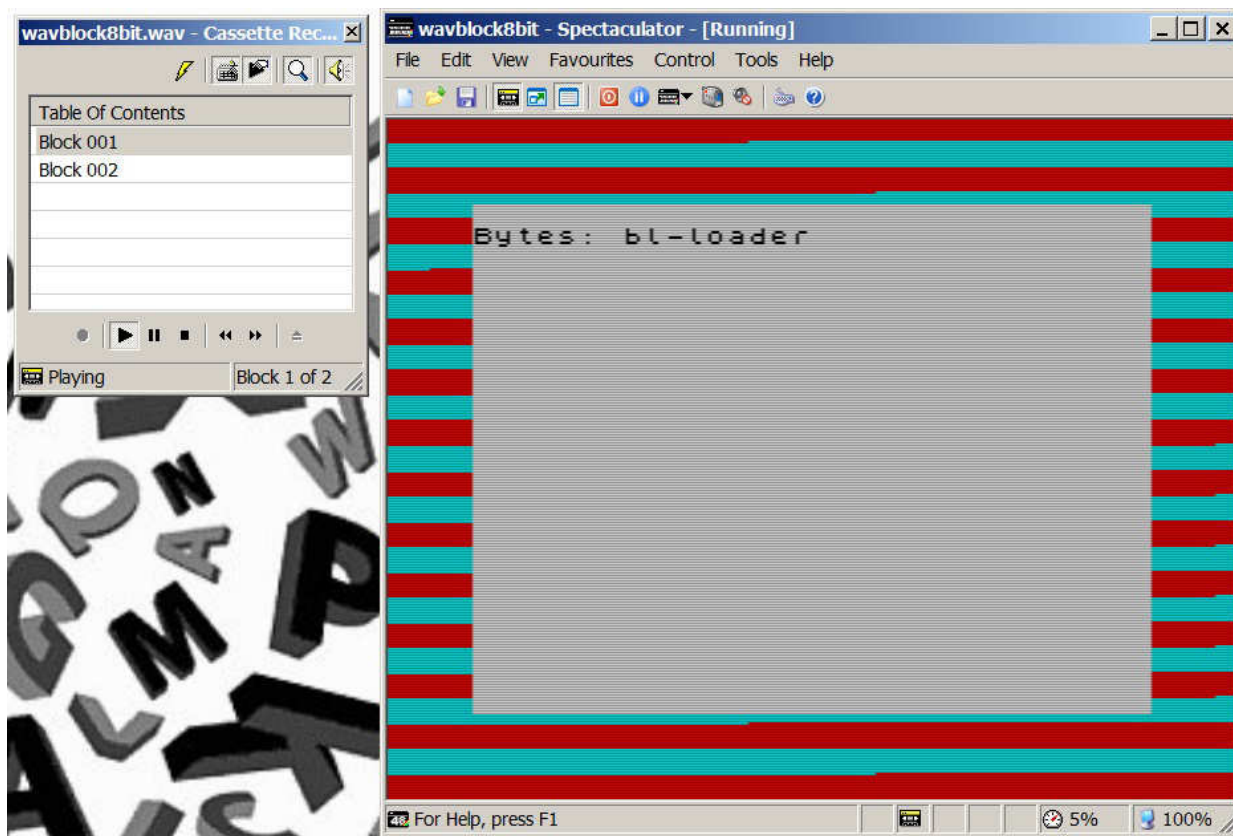


Рис. 6310. Spectaculator. Загрузка блока данных из *.wav файла, методом деления на куски.

Однако заметим, что полосы пилот-тона по экрану побежали вверх очень быстро, а звук сигнала тоньше. Вспомним, как при модернизации команды **БЕЕР** видели похожую ситуацию, когда изменяя частоту звука можно управлять движением полосок по рамке. После загрузки, компьютер выдаст **OK**, **: 1:**

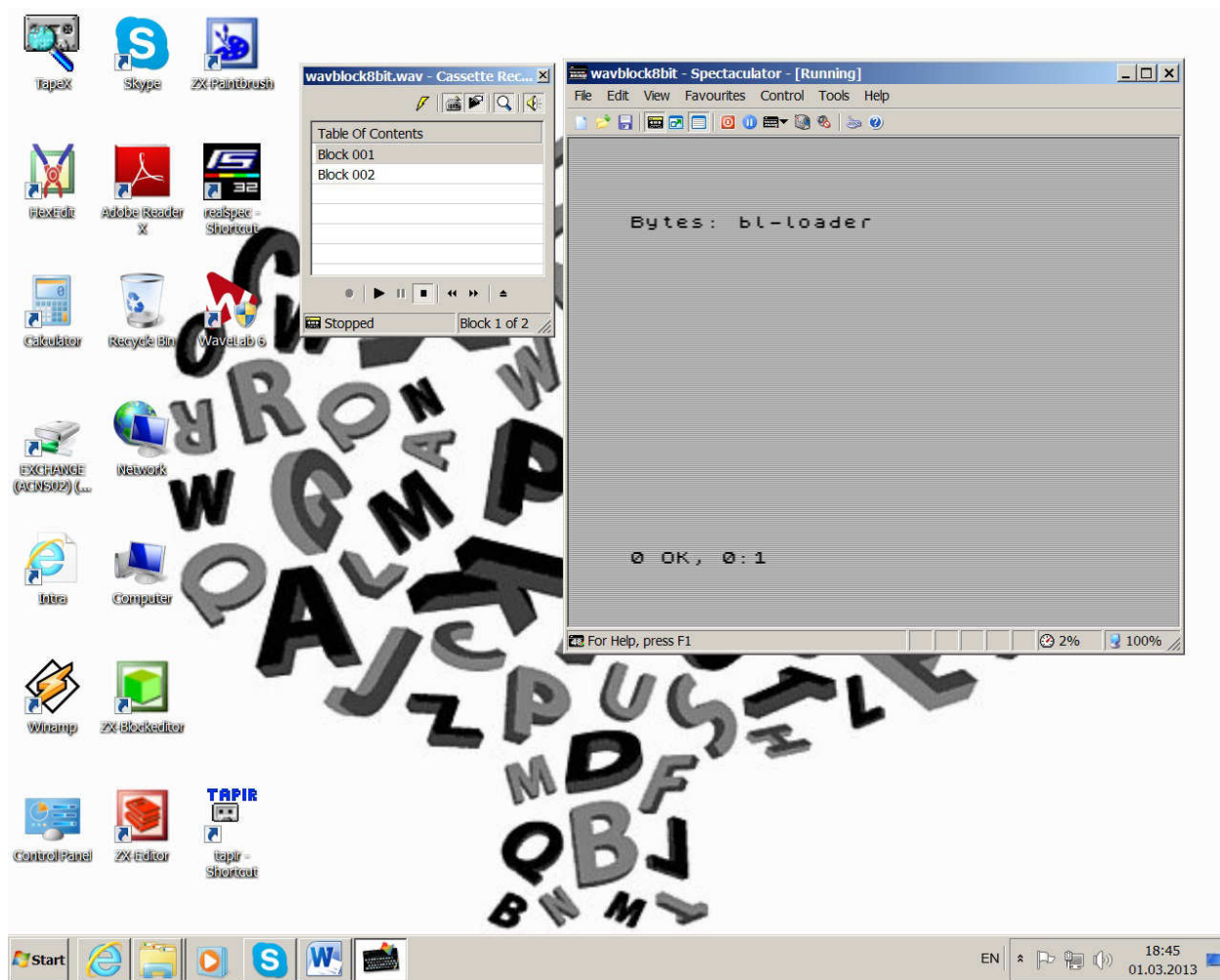


Рис. 6311. Успешное завершение считывание данных, созданных в звуковом редакторе.

Наберите `RANDOMIZE USA 30000`. Рамка станет фиолетовой, показав, что звуковой файл успешно считался, и превратился в машинную программу:

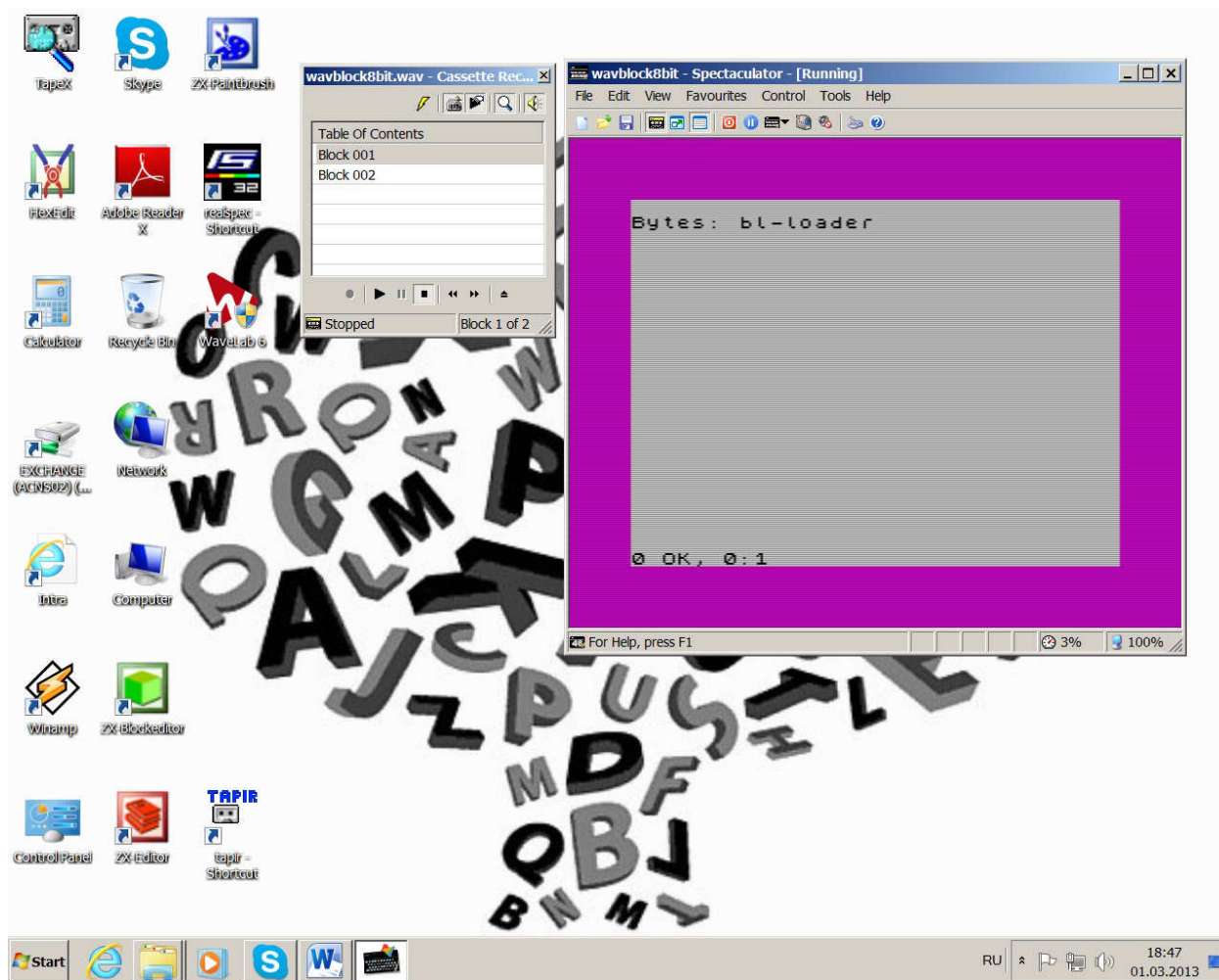


Рис. 6312. Spectaculator. Выполнение программы в машинных кодах из созданного звукового файла.

Глава 4.

Ручное изменение формы сигнала и его влияние на цветные полосы при загрузке.

Краткое содержание: создание сигнала пилот тона разной формы и частоты, исследование влияния формы сигнала на загрузку.

Поняв, как устроен сигнал, попробуем поэкспериментировать с его формой, и частотой. Создадим несколько разных видов сигнала пилот-тона и посмотрим, как это отразится на приеме сигнала и направлению движения полосок по рамке. Вспомним, как скруглены верхушки у сигналов, созданных программой TAPIR. Создадим 7 разных сигналов пилот-тона:

- 1) Со скругленным левым верхним и правым нижним краями.
- 2) Со скругленным правым верхним и левым нижним краями.
- 3) Со скругленным левым верхним краем.
- 4) С уменьшенным количеством сэмплов верхней части 26/27.
- 5) С уменьшенным количеством сэмплов нижней части 27/26.
- 6) С увеличенным количеством сэмплов верхней части 28/27
- 7) С увеличенным количеством верхней и нижней части 28/28

Откроем «*pilot.wav*», создадим новый файл, и скопируем его туда. Карандашом «сточим» левый верхний и правый нижний край. Размножим полученный фрагмент, чтобы получился сигнал в районе 3 секунд. Сохраните файл под именем «*pilotsign1.wav*»:

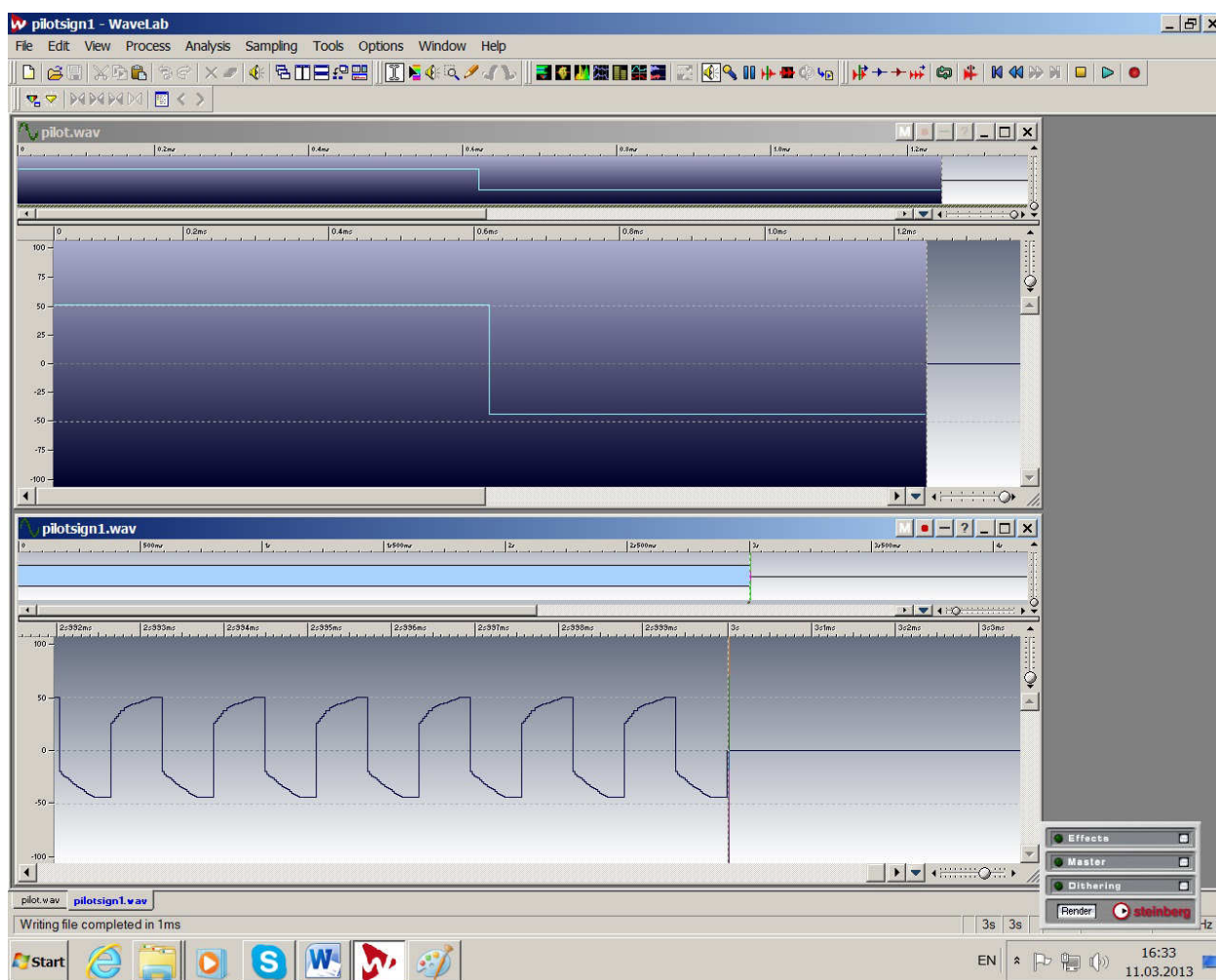


Рис. 6400. Редактор Wavelab: Создание экспериментального сигнала пилот-тона со сточенными краями.

Теперь сделаем сигнал, скруглив противоположные края. Для этого создайте еще один новый файл, точно также скопируйте туда «*pilot.wav*». Скруглите правый верхний и левый нижний край карандашом и размножьте фрагмент, чтобы получился однородный сигнал 3 секунды. Сохраните его, по имени пункта 2, «*pilotsign2.wav*»:

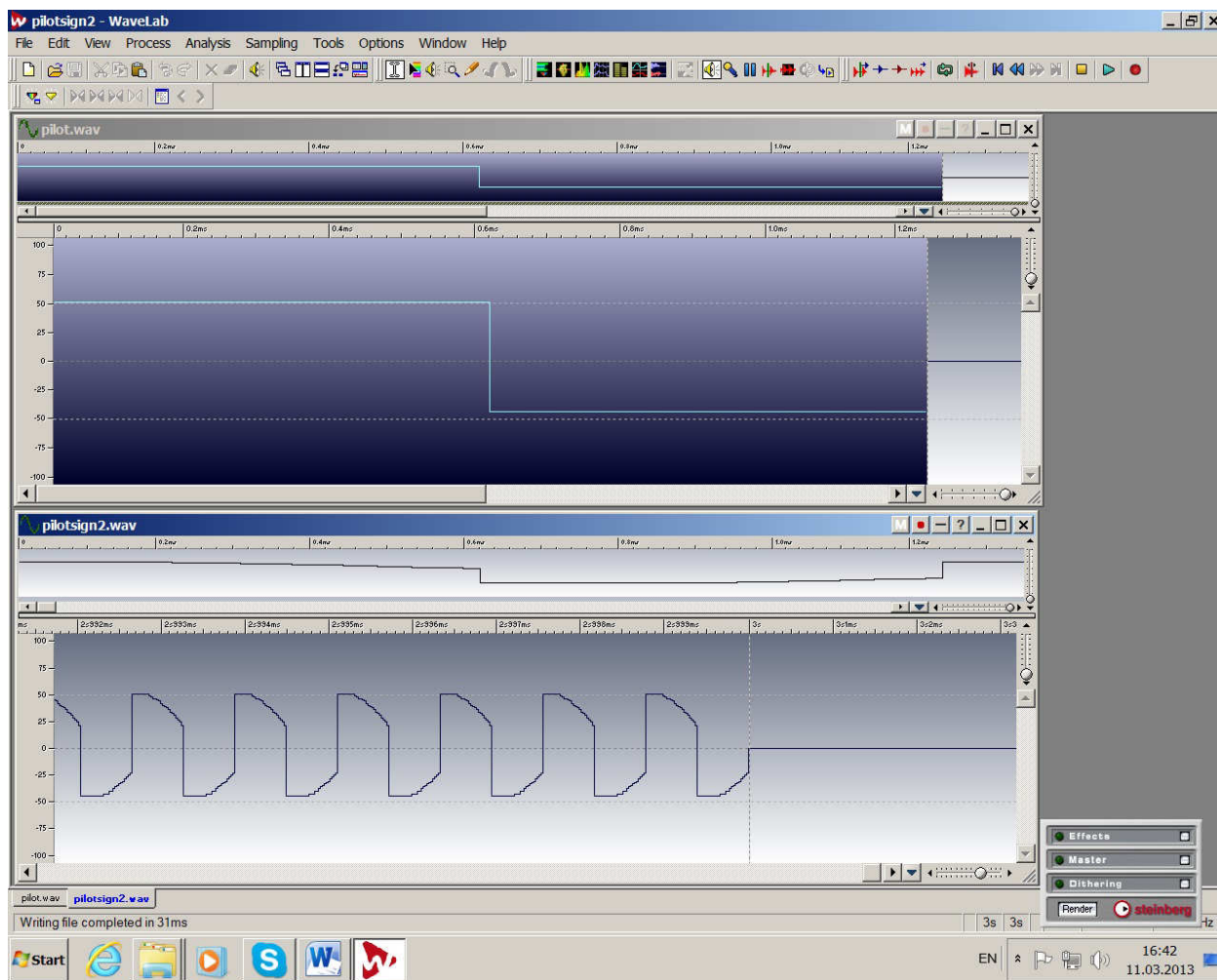


Рис. 6401. Редактор Wavelab: Создание экспериментального сигнала пилот-тона со сточенными краями.

Аналогичным образом создайте сигнал под пунктом 3, скруглив только левую верхнюю часть, оставив нижнюю, прямоугольной. Сохраните его под именем «*pilotsign3.wav*»:

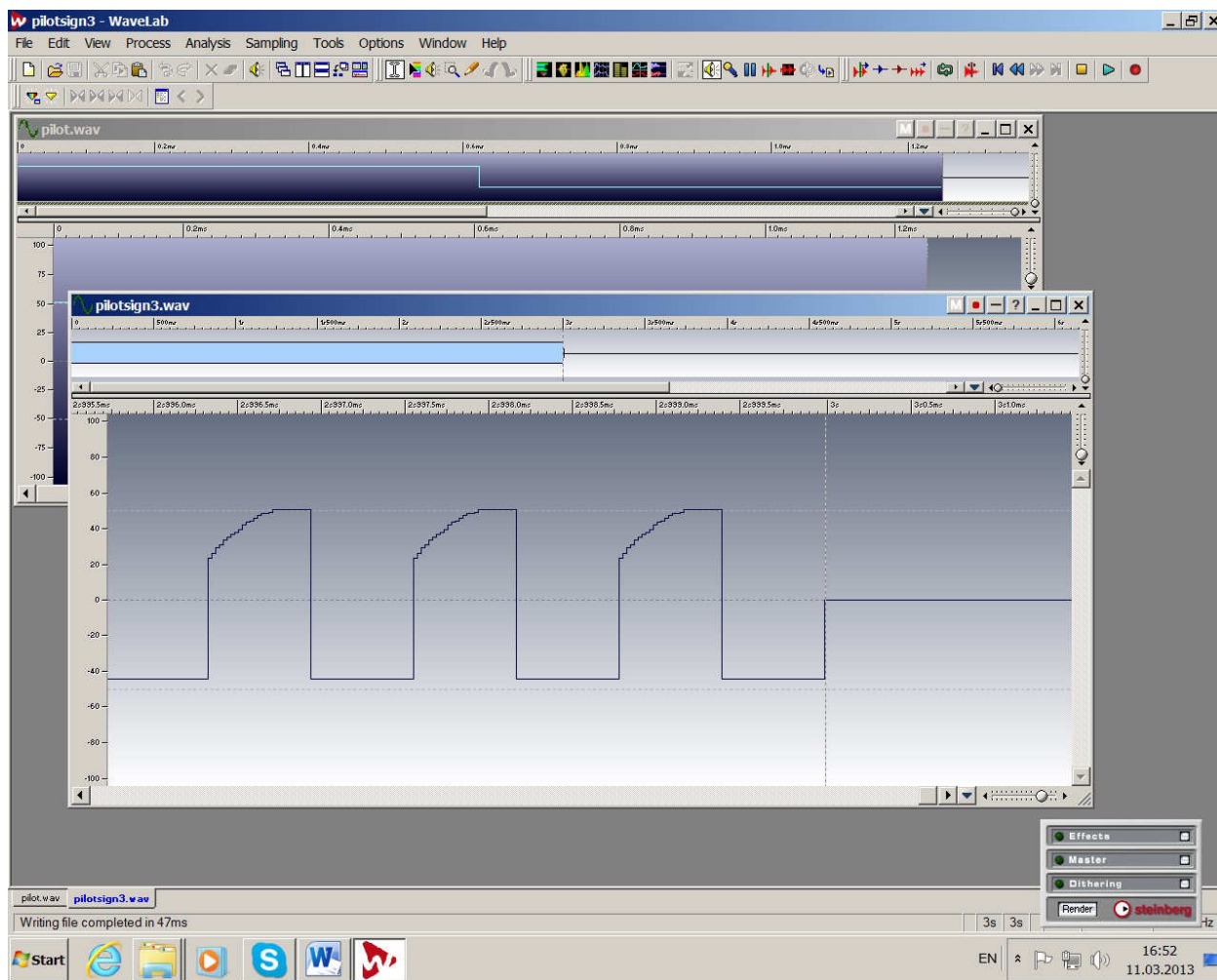


Рис. 6402. Wavelab: Создание экспериментального сигнала пилот-тона со сточенным левым краем.

Далее создайте два новых файла «Untitled1» и «Untitled2», и снова скопируйте в них «pilot.wav». В первом файле «выкусите» сэмплы из верхней части, чтобы она получилось на один неделимый фрагмент короче, чем нижняя (26 сэмплов). Во втором файле, в нижней части уберите сэмплы из сигнала, чтобы нижний участок был короче верхнего. Выглядеть они будут вот так:

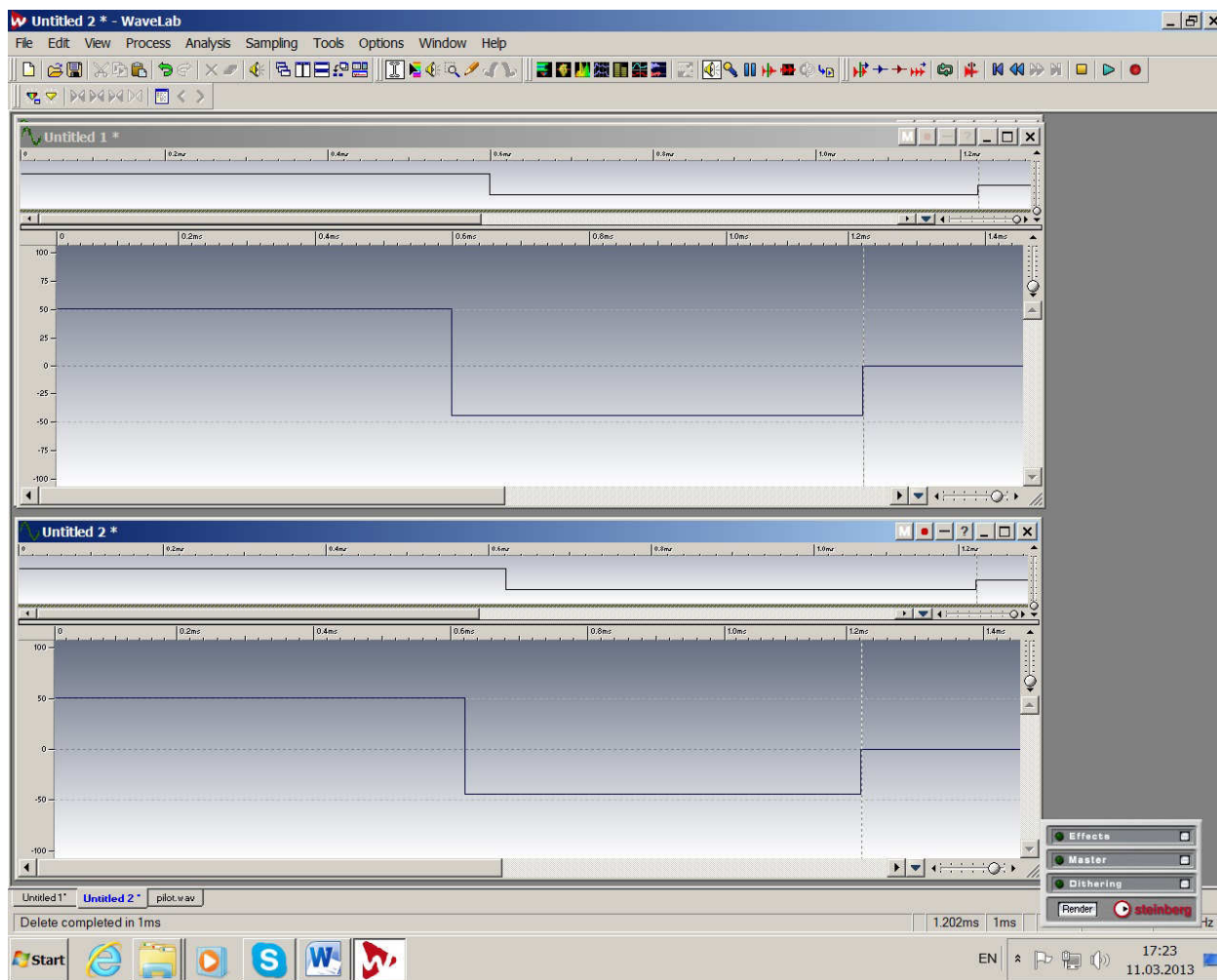


Рис. 6403. Wavelab: Создание заготовок несимметричных уменьшенных сигналов пилот-тона.

Размножьте эти несимметричные фрагменты до получения сигналов, длиной 3 секунды каждый, и сохраните под именами «*pilotsign4.wav*» «*pilotsign5.wav*»:

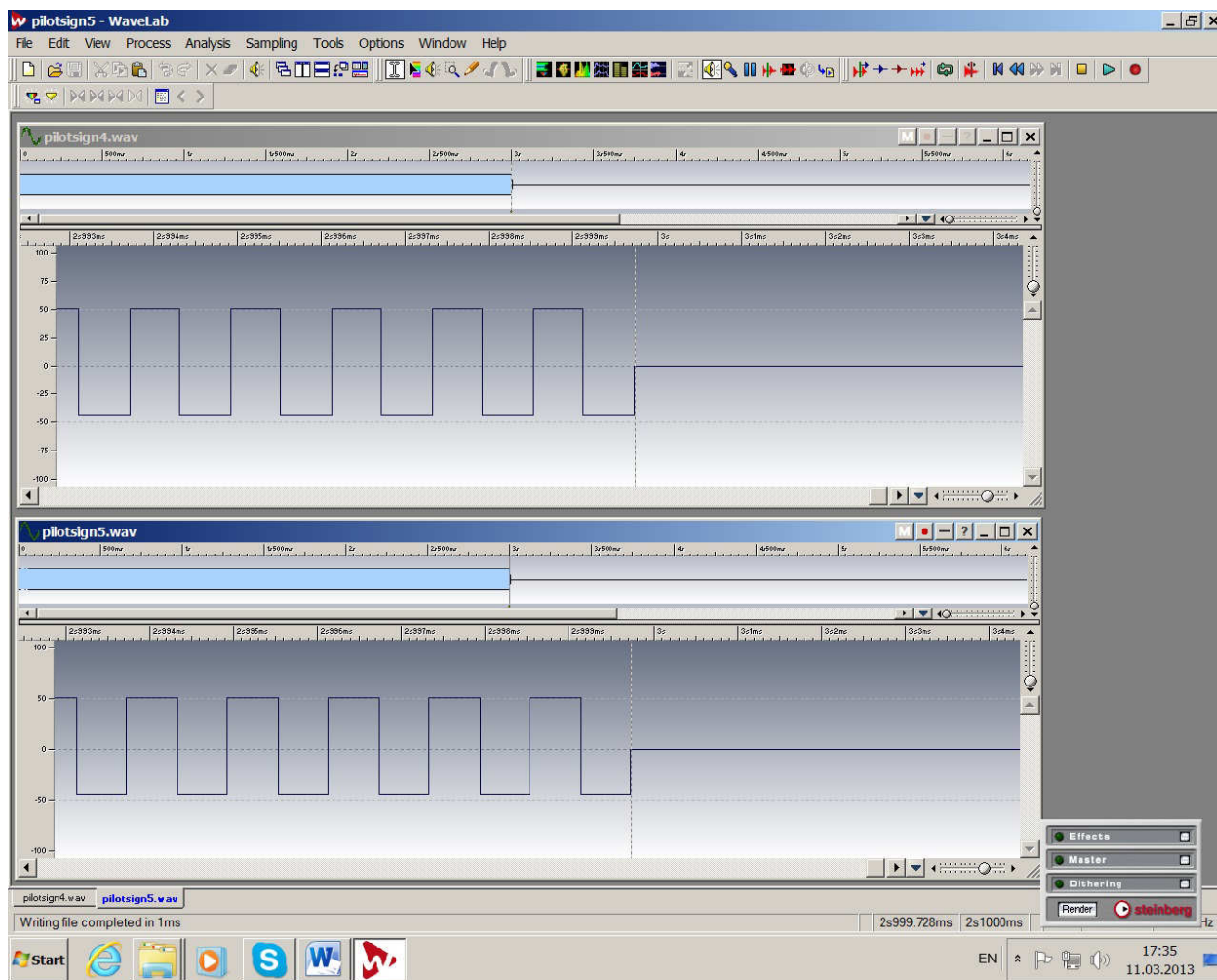


Рис. 6404. Wavelab: Готовые уменьшенные несимметричные сигналы экспериментального пилот-тона.

Аналогично создадим два заключительных вида сигнала с дополнительными сэмплами. Создайте два пустых файла, перекопируйте туда «*pilot.wav*». В первый добавьте дополнительный 28-й семпл в верхнюю часть, а во второй по семплу в верхнюю и нижнюю, чтобы получилось 28 и 28 семплов вверх и вниз. Сохраните фрагменты под именами «*pilotsign6.wav*» и «*pilotsign7.wav*»:

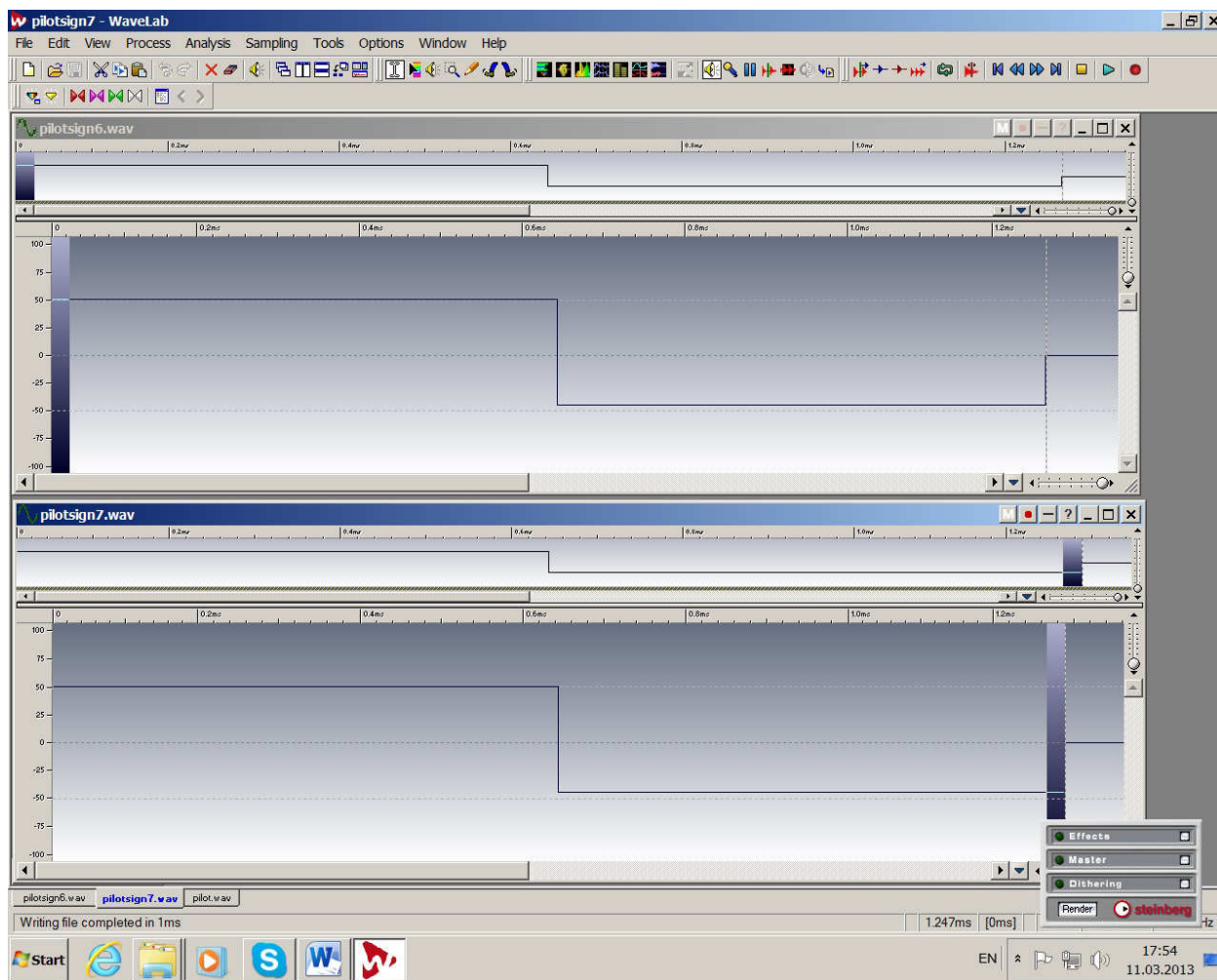


Рис. 6405. Создание заготовок увеличенных сигналов пилот-тона (симметричных и несимметричных).

Создайте из них сигналы по 3 секунды каждый:

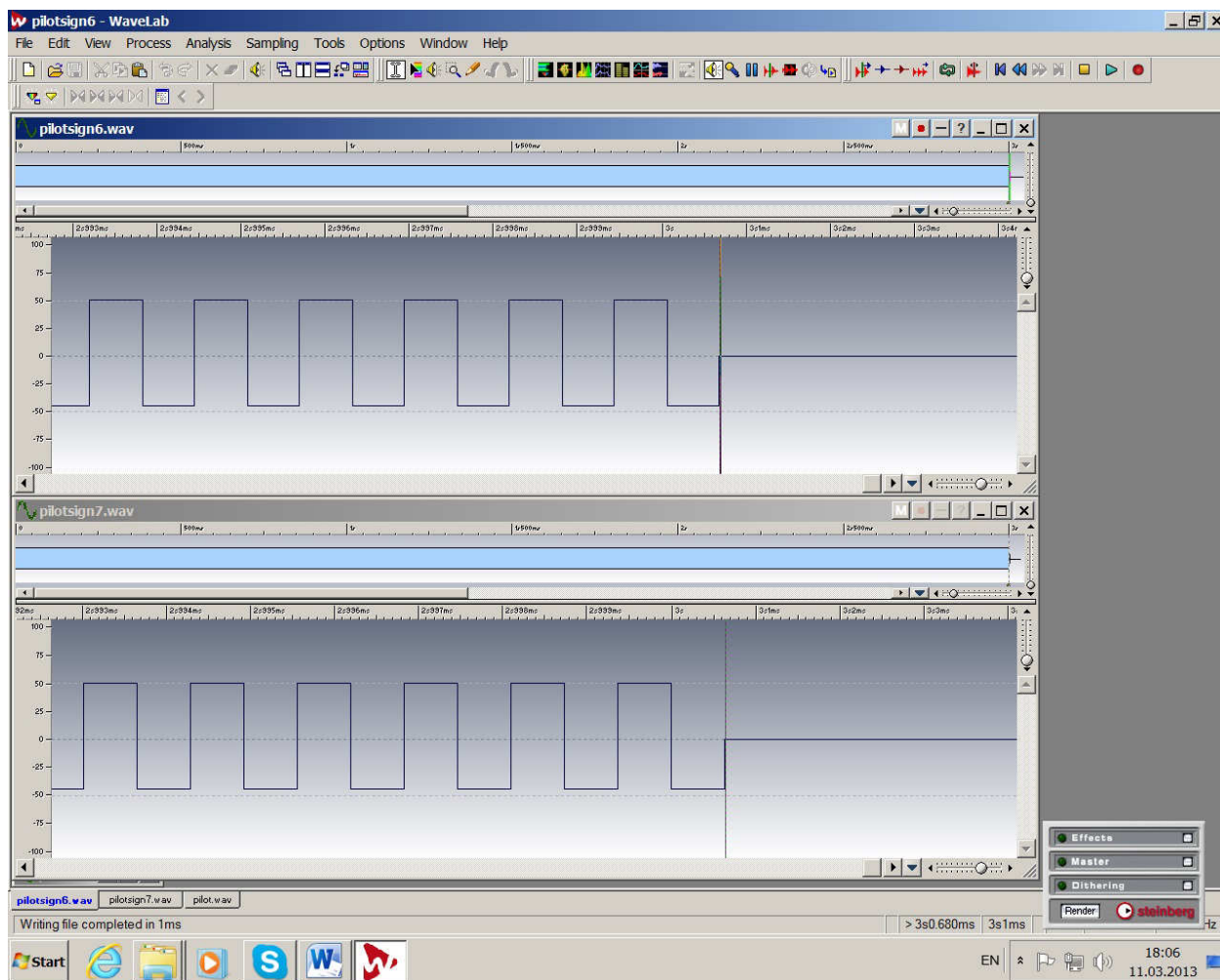


Рис. 6406. Wavelab: Готовые увеличенные несимметричные и несимметричные сигналы пилот-тона.

Окончательно сохраните сигналы, и выйдите из Wavelab. Создав сигналы разной формы, будем их проверять, а при звучании внимательно смотреть на полосы, бегущие по рамке. Откройте Spectaculator, и наберите `LOAD` `ENTER`. Теперь поочередно добавляем файлы в виртуальный магнитофон, слушаем сигнал, и смотрим на цветные полосы.

От файлов «*pilotsign1.wav*», «*pilotsign2.wav*» и «*pilotsign3.wav*» полосы по рамке бегут крайне быстро. Полоски бегут с одинаковой скоростью, на слух почти идентичные. Из этого можно сделать вывод, что скругления краев на сигнал загрузки не влияет. По крайней мере визуально.

Загрузив «хромые» «*pilotsign4.wav*» и «*pilotsign5.wav*» сразу увидите и услышите разницу. Звук стал тоньше, полосы двигаются вниз, а их скорость стала значительно меньше. Следовательно, на скорость движения влияет именно размерность элемента, формирующая сигнал определенной частоты. Но компьютеру без разницы, какая часть сигнала больше, верхняя или нижняя.

Теперь включим «*pilotsign6.wav*» с увеличенным сигналом. Звук стал, более низким и полосы тихо поползли вверх. Файл «*pilotsign7.wav*» звучит еще ниже, а полосы, хоть и не очень быстро, но снова побежали вниз.

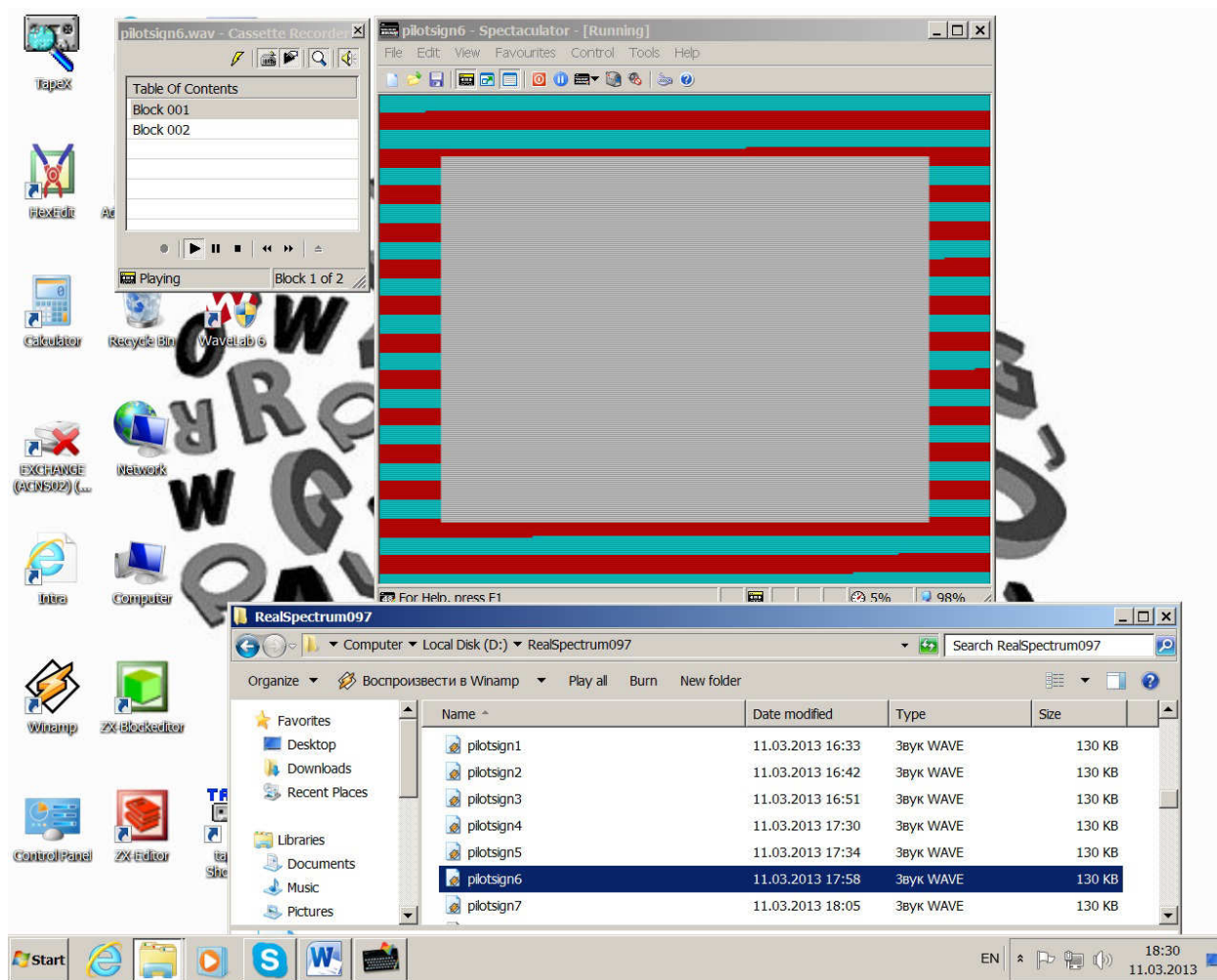


Рис. 6407. Звуковой файл «pilotsign6.wav» в процессе тестирования на эмуляторе.

Во всех случаях сигнал корректно считался, о чем символизировали красно-голубые полосы на рамке эмулятора.

Глава 5. Создание *.tar файлов из звуковых данных.

Краткое содержание: создание tar файла из звукового, работа с программой обработки магнитофонных файлов.

До этого момента много говорилось о том, как создать звуковой файл. А теперь попробуем создать из нашего звукового файла «wavblock8bit.wav» (часть 6 глава 3), создать *.tar файл. Не так давно была обнаружена специальная программа, разработанная российским любителем спектрума, которая называется «Обработка магнитофонных файлов ZX-Spectrum V1.0». Она все делает автоматически, и поэтому очень удобна в работе:

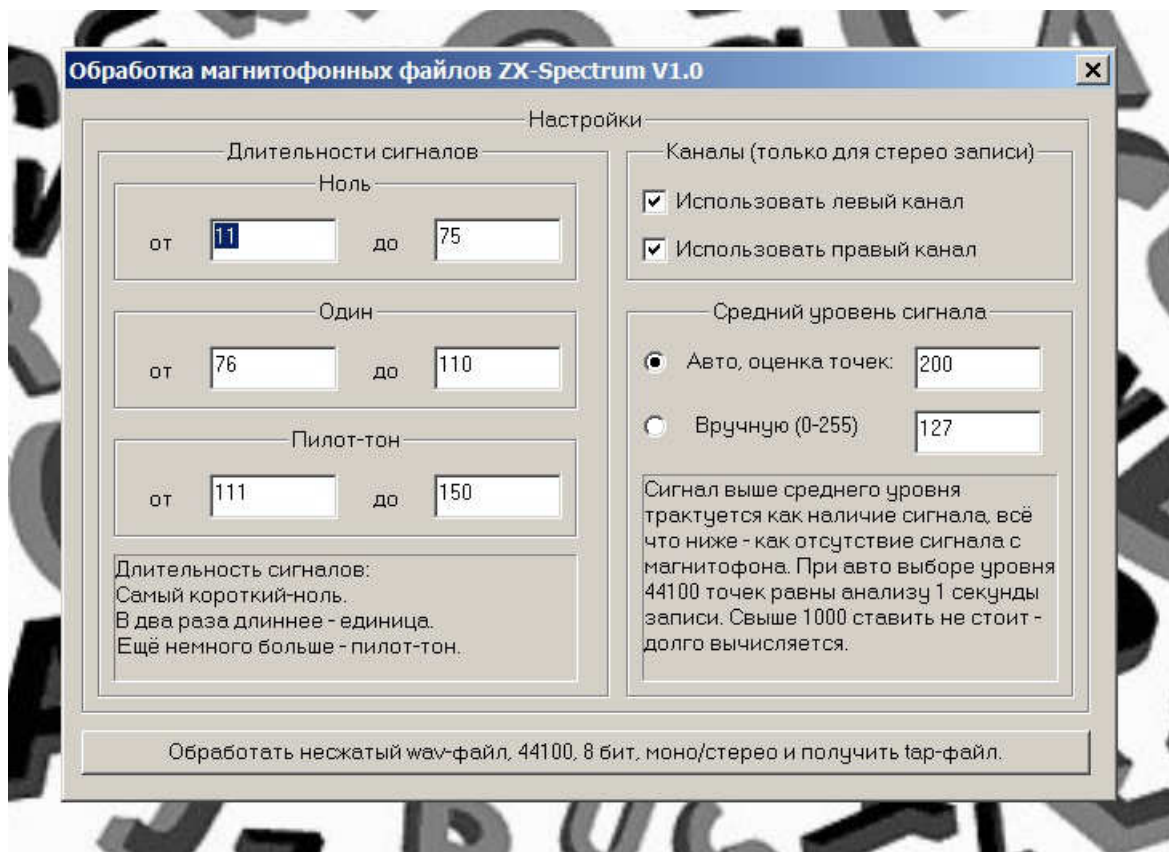


Рис. 6500. Общий вид программы обработки магнитофонных файлов ZX-Spectrum V1.0

Для того, чтобы преобразовать звуковой файл в *.tar нажмите длинную кнопку внизу окна. Откроется дополнительное окно «Выбор звукового файла для расшифровки». Выберите в нем «wavblock8bit.wav»:

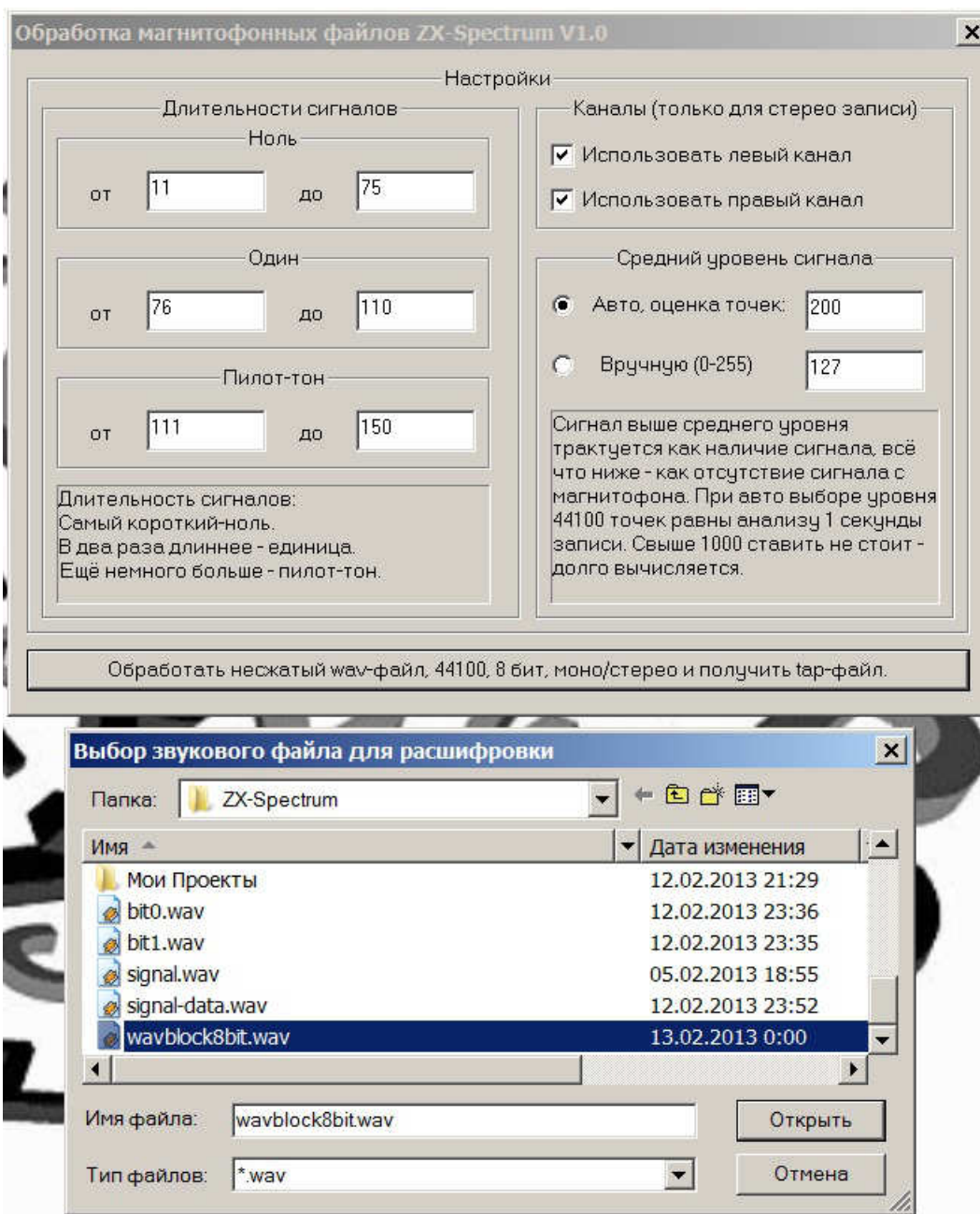


Рис. 6501. Окно «Выбор звукового файла для расшифровки». Открытие .wav файла.

После нажатия, окно закроется и выскочит информационное окно «Сообщение», об успешном создании «wavblock8bit.tap» и «wavblock8bit.txt» файла:

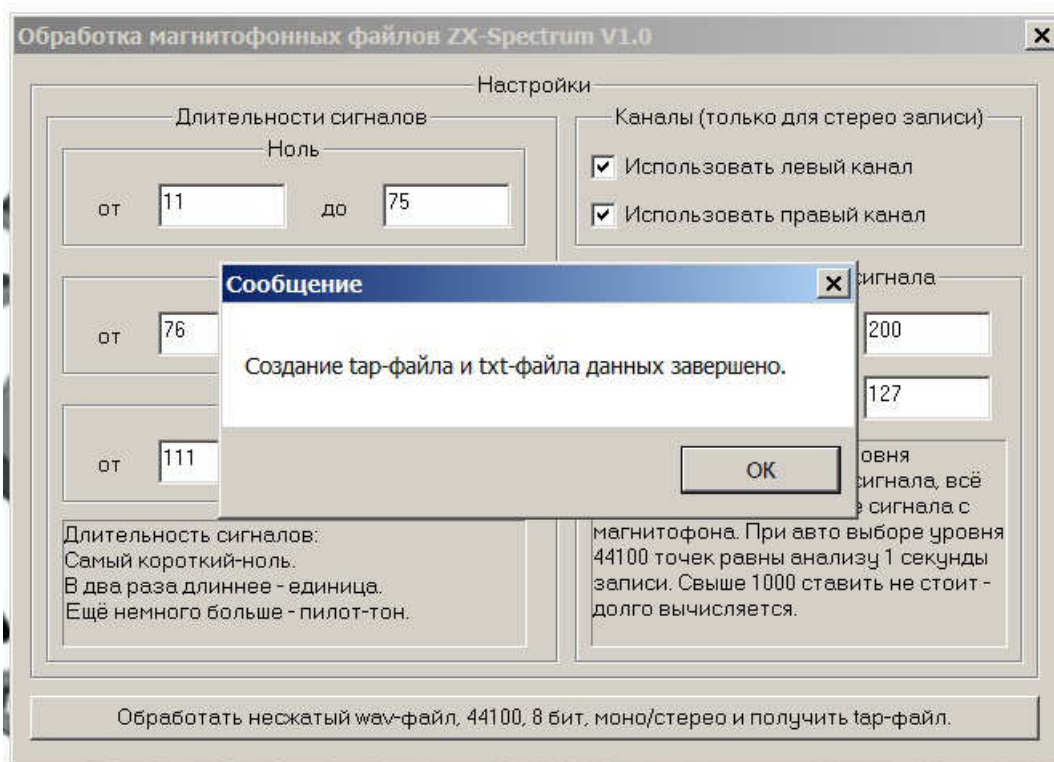


Рис. 6502. Информационное окно. Успешное преобразование «wavblock8bit.wav» в «wavblock8bit.tap».

Закрывайте окно, выходите из программы. Файл «wavblock8bit.tap» создан. Вместе с ним в комплекте появился текстовый файл отчета с распечаткой программы:

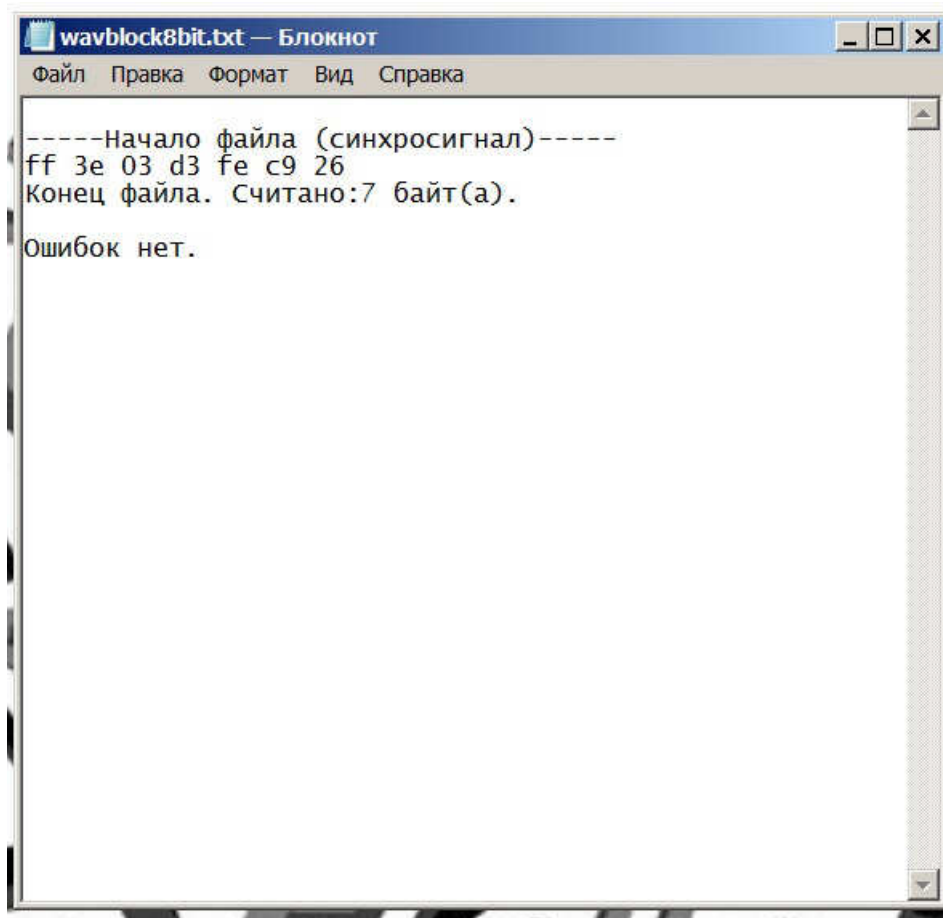


Рис. 6503. Текстовый файл с отчетом и кодом программы.

Таким образом создали *.tap файл из собственноручно созданного в музыкальном редакторе звукового файла. Полученный файл можно проверить в эмуляторе.

ПРИЛОЖЕНИЯ.

Приложение 1.

Таблица экономичных эквивалентов чисел для использования в программах.

В большинстве программ-загрузчиков на BASIC, часто можно увидеть такую строку:

BORDER x: PAPER x: INK x: CLS

где x – число от 0 до 7.

Числовой эквивалент, в таком случае, занимает целых 7 байт в памяти (без команд и номера строки). Поэтому в некоторых случаях можно увидеть, например BORDER VAL "1". Такое написание существенно сократит программу. В этом случае, запись числа в памяти займет всего 4 байта. Но можно добиться и еще более совершенного варианта, если использовать функции. В таблице приведены наиболее оптимальные варианты-заменители чисел, которые можно использовать в BASIC программах. Дробных значений цветов, которые получаются при вычислениях, не бывает и компьютер округляет до ближайшего целого.

Выражение	Числовой эквивалент	Итоговый цвет	Количество байт в BASIC программе	Экономия байт по сравнению с числом
NOT PI	0	Черный	2	5
SGN PI	1	Синий	2	5
LN PI	1,1447299	Синий	2	5
ATN PI	1,2626273	Синий	2	5
SQR PI	1,7724539	Красный	2	5
PI	3,1415927	Фиолетовый	1	6
PI+SGN PI	4,1415927	Зеленый	4	3
SQR EXP PI	4,8104774	Голубой	3	4
EXP SQR PI	5,8852772	Желтый	3	4
PI+PI	6,2831853	Желтый	3	4
EXP PI/PI	7,3659112	Белый	4	3

Приложение 2.

Полный набор русских символов.

В части 3 главе 2 описывался метод русификации ПЗУ. В приложении приведен набор данных для полной русификации знакогенератора, практически по всем стандартам соответствия, кроме пары букв. От себя дополнительно добавил маленькую букву «ё» вместо символа «&», а также не стал трогать символы «\$» и «#», так как они используются в «BASIC».

Шрифт можно вставлять как в ПЗУ, так и в ОЗУ, с соответствующим изменением адреса размещения в переменной **CHARS**. Для размещения в ПЗУ нужно преобразовать текст в данные с помощью эмулятора, как описывалось ранее.

DEFB 0, 0, 0, 0, 0, 0, 0, 0 ;

пробел

DEFB 0, 16, 16, 16, 16, 0, 16, 0 ;	!	
DEFB 0, 36, 36, 0, 0, 0, 0, 0 ;	..	
DEFB 0, 36, 126, 36, 36, 126, 36, 0 ;	#	
DEFB 0, 8, 62, 40, 62, 10, 62, 8 ;	\$	
DEFB 0, 98, 100, 8, 16, 38, 70, 0 ;	%	
DEFB 40, 0, 56, 68, 120, 64, 60, 0 ;	& =	ë
DEFB 0, 8, 16, 0, 0, 0, 0, 0 ;	,	
DEFB 0, 4, 8, 8, 8, 8, 4, 0 ;	(
DEFB 0, 32, 16, 16, 16, 16, 32, 0 ;)	
DEFB 0, 0, 20, 8, 62, 8, 20, 0 ;	*	
DEFB 0, 0, 8, 8, 62, 8, 8, 0 ;	+	
DEFB 0, 0, 0, 0, 0, 8, 8, 16 ;	,	
DEFB 0, 0, 0, 0, 62, 0, 0, 0 ;	-	
DEFB 0, 0, 0, 0, 0, 24, 24, 0 ;	.	
DEFB 0, 0, 2, 4, 8, 16, 32, 0 ;	/	
DEFB 0, 60, 70, 74, 82, 98, 60, 0 ;	0	
DEFB 0, 24, 40, 8, 8, 8, 62, 0 ;	1	
DEFB 0, 60, 66, 2, 60, 64, 126, 0 ;	2	
DEFB 0, 60, 66, 12, 2, 66, 60, 0 ;	3	
DEFB 0, 8, 24, 40, 72, 126, 8, 0 ;	4	
DEFB 0, 126, 64, 124, 2, 66, 60, 0 ;	5	
DEFB 0, 60, 64, 124, 66, 66, 60, 0 ;	6	
DEFB 0, 126, 2, 4, 8, 16, 16, 0 ;	7	
DEFB 0, 60, 66, 60, 66, 66, 60, 0 ;	8	
DEFB 0, 60, 66, 66, 62, 2, 60, 0 ;	9	
DEFB 0, 0, 0, 16, 0, 0, 16, 0 ;	:	
DEFB 0, 0, 16, 0, 0, 16, 16, 32 ;	;	
DEFB 0, 0, 4, 8, 16, 8, 4, 0 ;	<	
DEFB 0, 0, 0, 62, 0, 62, 0, 0 ;	=	
DEFB 0, 0, 16, 8, 4, 8, 16, 0 ;	>	
DEFB 0, 60, 66, 4, 8, 0, 8, 0 ;	?	
DEFB 0, 156, 162, 226, 162, 162, 156, 0 ;	@ =	Ю
DEFB 0, 120, 132, 132, 252, 132, 132, 0 ;	A =	А
DEFB 0, 248, 128, 248, 132, 132, 248, 0 ;	B =	Б
DEFB 0, 132, 132, 132, 132, 132, 254, 2 ;	C =	Ц
DEFB 0, 60, 68, 68, 68, 68, 254, 130 ;	D =	Д
DEFB 0, 252, 128, 248, 128, 128, 252, 0 ;	E =	Е
DEFB 0, 124, 146, 146, 146, 124, 16, 0 ;	F =	Ф
DEFB 0, 124, 64, 64, 64, 64, 64, 0 ;	G =	Г
DEFB 0, 132, 72, 48, 48, 72, 132, 0 ;	H =	Х
DEFB 0, 132, 140, 148, 164, 196, 132, 0 ;	I =	И
DEFB 48, 132, 140, 148, 164, 196, 132, 0 ;	J =	Я
DEFB 0, 136, 144, 224, 144, 136, 132, 0 ;	K =	К
DEFB 0, 60, 68, 68, 68, 68, 132, 0 ;	L =	Л
DEFB 0, 132, 204, 180, 132, 132, 132, 0 ;	M =	М
DEFB 0, 132, 132, 252, 132, 132, 132, 0 ;	N =	Н
DEFB 0, 120, 132, 132, 132, 132, 120, 0 ;	O =	О
DEFB 0, 252, 132, 132, 132, 132, 132, 0 ;	P =	П
DEFB 0, 124, 132, 132, 124, 68, 132, 0 ;	Q =	Я
DEFB 0, 248, 132, 132, 248, 128, 128, 0 ;	R =	Р
DEFB 0, 120, 132, 128, 128, 132, 120, 0 ;	S =	С
DEFB 0, 254, 16, 16, 16, 16, 16, 0 ;	T =	Т

DEFB 0, 132, 132, 132, 124, 4, 120, 0 ;	U = У
DEFB 0, 146, 84, 56, 56, 84, 146, 0 ;	U = Ж
DEFB 0, 248, 132, 248, 132, 132, 248, 0 ;	W = В
DEFB 0, 128, 128, 248, 132, 132, 248, 0 ;	X = Ъ
DEFB 0, 130, 130, 242, 138, 138, 242, 0 ;	Y = Ы
DEFB 0, 120, 132, 24, 4, 132, 120, 0 ;	Z = Э
DEFB 0, 146, 146, 146, 146, 146, 254, 0 ;	[= Ш
DEFB 0, 120, 132, 28, 4, 132, 120, 0 ;	\ = Э
DEFB 0, 146, 146, 146, 146, 146, 255, 1 ;] = Щ
DEFB 0, 132, 132, 132, 124, 4, 4, 0 ;	^ = Ч
DEFB 0, 192, 64, 124, 66, 66, 124, 0 ;	_ = Ъ
DEFB 0, 0, 76, 82, 114, 82, 76, 0 ;	ѐ = Ю
DEFB 0, 0, 56, 4, 60, 68, 60, 0 ;	а = а
DEFB 0, 0, 120, 64, 120, 68, 120, 0 ;	б = б
DEFB 0, 0, 68, 68, 68, 68, 126, 2 ;	с = ц
DEFB 0, 0, 28, 36, 36, 36, 126, 66 ;	д = д
DEFB 0, 0, 56, 68, 120, 64, 60, 0 ;	е = е
DEFB 0, 0, 56, 84, 84, 84, 56, 16 ;	ф = ф
DEFB 0, 0, 60, 32, 32, 32, 32, 0 ;	г = г
DEFB 0, 0, 68, 40, 16, 40, 68, 0 ;	h = х
DEFB 0, 0, 68, 76, 84, 100, 68, 0 ;	i = и
DEFB 0, 16, 68, 76, 84, 100, 68, 0 ;	j = й
DEFB 0, 0, 68, 72, 112, 72, 68, 0 ;	k = к
DEFB 0, 0, 28, 36, 36, 36, 68, 0 ;	l = л
DEFB 0, 0, 68, 108, 84, 68, 68, 0 ;	m = м
DEFB 0, 0, 68, 68, 124, 68, 68, 0 ;	n = н
DEFB 0, 0, 56, 68, 68, 68, 56, 0 ;	о = о
DEFB 0, 0, 124, 68, 68, 68, 68, 0 ;	p = п
DEFB 0, 0, 60, 68, 68, 60, 68, 0 ;	q = я
DEFB 0, 0, 120, 68, 68, 120, 64, 0 ;	г = р
DEFB 0, 0, 56, 68, 64, 68, 56, 0 ;	s = с
DEFB 0, 0, 124, 16, 16, 16, 16, 0 ;	t = т
DEFB 0, 0, 68, 68, 60, 4, 56, 0 ;	u = у
DEFB 0, 0, 84, 56, 16, 56, 84, 0 ;	v = ж
DEFB 0, 0, 120, 68, 120, 68, 120, 0 ;	w = в
DEFB 0, 0, 64, 64, 120, 68, 120, 0 ;	x = ъ
DEFB 0, 0, 66, 66, 114, 74, 114, 0 ;	y = ы
DEFB 0, 0, 56, 68, 24, 68, 56, 0 ;	z = э
DEFB 0, 0, 68, 84, 84, 84, 124, 0 ;	{ = ш
DEFB 0, 0, 56, 68, 28, 68, 56, 0 ;	= э
DEFB 0, 0, 68, 84, 84, 84, 126, 2 ;	} = щ
DEFB 0, 0, 68, 68, 60, 4, 4, 0 ;	↑ = ч
DEFB 0, 0, 96, 32, 60, 34, 60, 0 ;	© = ъ

СОДЕРЖАНИЕ:

ЧАСТЬ 1: Настройка эмуляторов и обзор ПО под windows.

Предисловие.....	
Глава 1. Обзор преимуществ и недостатков эмуляторов.....	
Глава 2. Настройка Spectaculator для загрузки с магнитной ленты в реальном времени.....	

ЧАСТЬ 2. Создание *.tap и *.tzh файлов эмуляторами на PC под windows.

Глава 1. Создание самозапускаемых блоков «Bytes:» с BASIC программой внутри.....	
--	--

Глава 2. Создание *.tzh файла с помощью эмулятора Realspectrum.....	
Глава 3. Создание символьного массива, начиненного машинной программой.....	
Глава 4. Запись автостартующего блока «Bytes:» с BASIC-строкой из машинного кода.....	
Глава 5. Создание и запись автозапускного блока кодов в машинный стек.....	
Глава 6. Загрузка машинной программы в область экрана.....	
Глава 7. Загрузка и запуск блоков с переходом через предел 65535.....	
Глава 8. Простейшая загрузка файла без заголовка с измененными цветами полос.....	
Глава 9. Запись картинки с нижними строками и длинным заголовком с управляющими кодами.	

ЧАСТЬ 3. Работы по модификации ПЗУ эмулятора ZX-Spectrum.

Глава 1. Настройка эмулятора для изменения данных ПЗУ.....	
Глава 2. Модернизация встроенного шрифта и текстовой информации в ПЗУ.....	
Глава 3. Изменение команд и сообщений интерпретатора BASIC.....	
Глава 4. Простая модификация программ в ПЗУ и изменение цвета полос загрузки.....	
Глава 5. Создание и запуск простейшей программы в ПЗУ.....	

ЧАСТЬ 4. Создание блоков данных утилитами под windows XP и 7.

Глава 1. Создание и запись картинки с нижними строками.....	
Глава 2. Создание простейшей BASIC программы без эмулятора.....	
Глава 3. Основы работы с блоками и заголовками магнитофонных файлов.....	
Глава 4. Создание звуковых .wav файлов с данными.....	
Глава 5. Создание блока кодов из фрагмента данных.....	
Глава 6. Создание искусственного блока заголовка.....	

ЧАСТЬ 5. Примеры создания сложных программ с помощью эмуляторов и утилит.

Глава 1. Создание длинного правильного блока «Program:» с невидимыми данными.....	
Глава 2. Загрузка BASIC программы блоком «Bytes:» и запуском через RANDOMIZE USR	
Глава 3. ZX-DESKTOP, или ностальгия по windows.....	
Глава 4. Создание картинки заголовками с коротким пилот-тоном и сохранением в память.	

ЧАСТЬ 6. Работа с сигналом и звуком загружаемых данных.

Глава 1. Создание мелодии из звукового сигнала загружаемых данных.....	
Глава 2. Исследование стандартного сигнала загрузки в звуковом редакторе.....	
Глава 3. Создание блока данных в звуковом редакторе.....	
Глава 4. Ручное изменение формы сигнала и его влияние на цветные полосы при загрузке.....	
Глава 5. Создание *.tap файлов из звуковых данных.....	

ПРИЛОЖЕНИЯ.

Приложение 1. Таблица экономических эквивалентов чисел для использования в программах.	
Приложение 2. Полный набор русских символов.....	