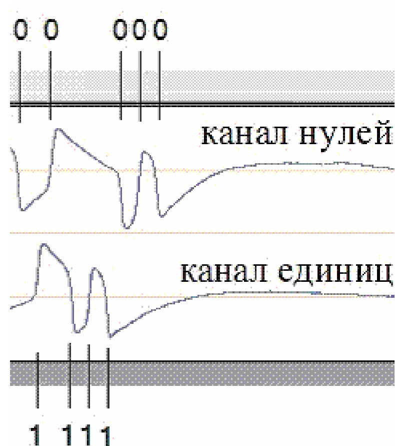
Процедуры раскодировки

Различаются два типа процедур раскодировки – «раскодировка 256-байтных блоков» (записанных командой типа SAVE R, об этом см. ниже, стр. 31–34), и «раскодировка программ произвольной длины» (записанных командой типа SAVE X, с этого начнём, см. стр. 1–30).

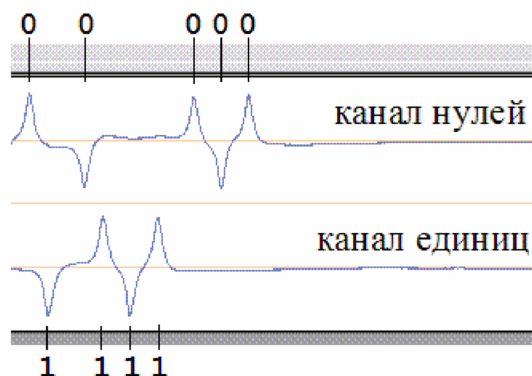
Раскодировка программы произвольной длины, записанной командой типа SAVE X

1. Сначала следует убедиться, что мы имеем дело с оцифровкой программы, записанной именно командой типа SAVE X. Для этого смотрим в аудиоредакторе график (waveform) конца записи – там должны быть видны перепады сигнала, характерные для завершающего байта $0512_{16} = 01011100_2$ с девятым битом чётности 0:



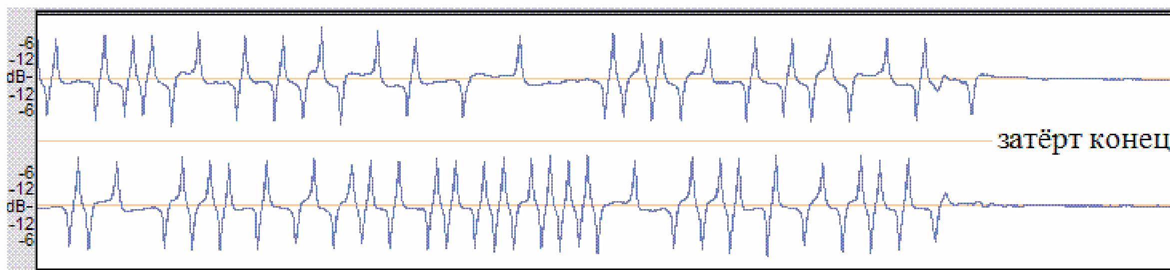
Так выглядит картина в оцифровке со стандартной АЧХ тракта воспроизведения бытового магнитофона: видны достаточно крутые передний и задний фронты «П-образных» импульсов. В процедуре раскодировки такие фронты будут детектироваться путём численного дифференцирования сигналов в каналах нулей и единиц (об этом речь пойдёт на стр. 26).

В оцифровках Виталия К., выполненных без коррекции АЧХ, картина несколько иная – вместо перепадов (фронтов) видны достаточно резкие «пики» Λ-образных импульсов:

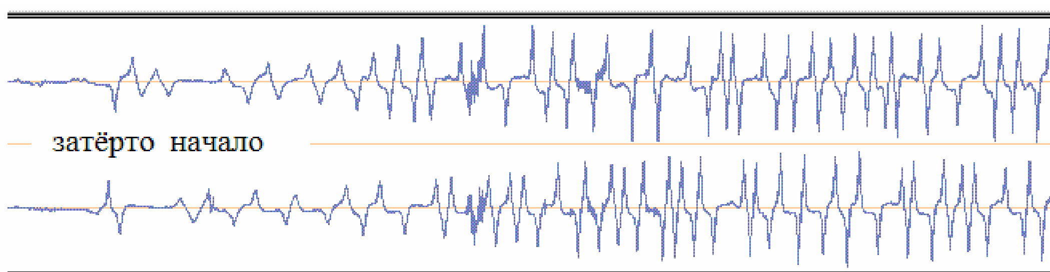


Такие сигналы уже не надо будет дифференцировать, их можно сразу возводить во вторую степень для получения однополярных импульсов, представляющих биты 0 и 1.

Если запись в оцифровке не имеет вида 256-байтных блоков (об их форме см. ниже, стр. 32) и при этом в конце она не содержит упомянутых девяти бит 010111000_2, то в такой записи, очевидно, испорчен конец, и её не удастся раскодировать:



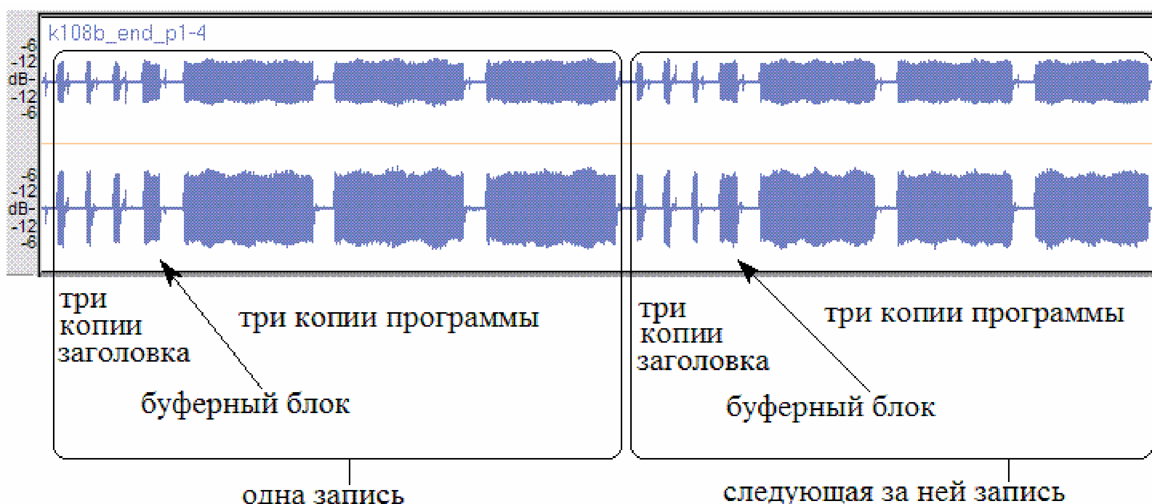
Иногда в оцифровках попадаются записи, у которых испорчено начало:



Пытаться раскодировать запись без начала так же бесполезно, как и запись без конца, если нет дополнительной информации или догадок о ней.

Таким образом, осмотр графиков сигнала в оцифровке – первый и очень важный этап. Его результаты нужны для выбора участка, пригодного к раскодированию. Если оцифровка содержит несколько пригодных копий, то выбирать следует ту из них, в которой стабильнее амплитуда импульсов, а искажения формы импульсов отсутствуют или хотя бы малы.

Эти же критерии выбора применимы и к «фортрано-подобным» записям; например, – к записям, сделанным в системах ОС ВТ МХТИ. Каждая запись, выполненная системой ВТ МХТИ-128, содержит разделённые небольшими паузами три копии короткого (21 байт) заголовка, буферный блок размером в 82 байта, и три копии основной части. Такая структура хорошо видна на графиках со сжатым масштабом по оси времени:

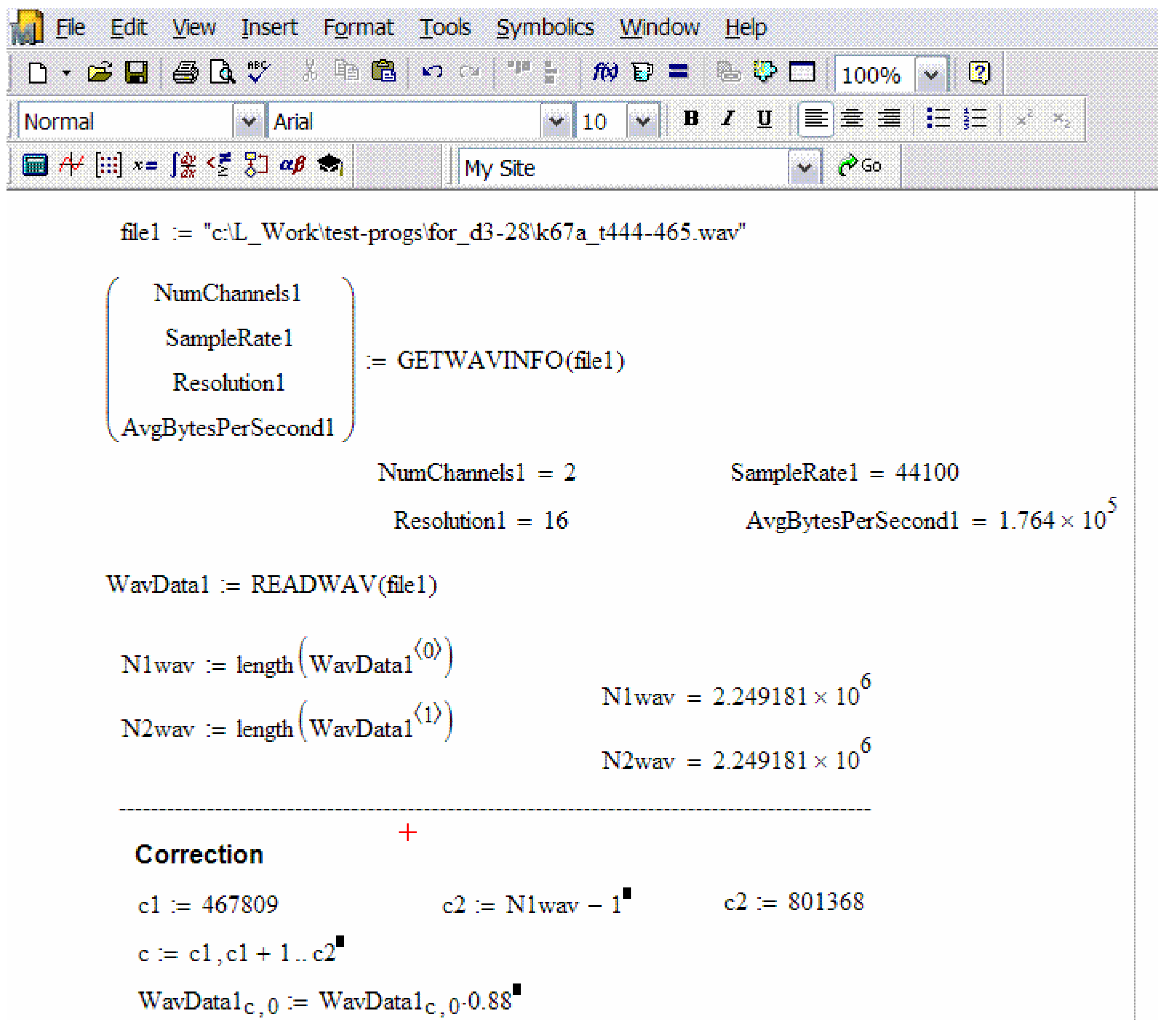


Каждый элемент записи (заголовок, буферный блок, и программа) раскодируется отдельно от остальных – как самостоятельный участок оцифровки. Выбранный для раскодирования участок следует записать в отдельный wav-файл, который затем будет прочитан раскодировщиком.

2. Раскодировщик у меня представляет собой составленный в программе Mathcad «документ» («рабочий лист», worksheet) с набором процедур. Можно работать и с несколькими «документами», по одному на каждую раскодируемую копию: если раскодировка идёт с трудом, то целесообразно заниматься двумя или более копиями параллельно, чтобы путём сравнения их результатов находить трудно уловимые сбойные места.

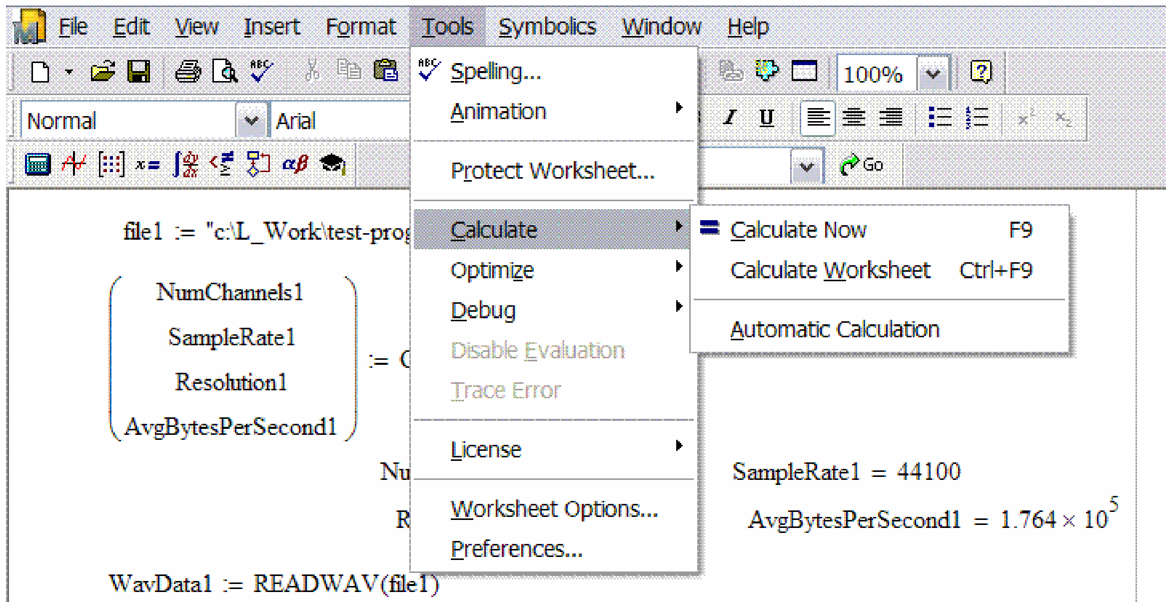
Программа Mathcad (Маткад) удобна тем, что в ней формулы набираются в стандартном для математической литературы виде, причём составление «документа» напоминает привычную работу с текстовым редактором. Можно по ходу дела копировать или перемещать фрагменты «документа» из одного его места в другое, удалять лишние строки, вставлять новые строки, набирать в них новые варианты процедур, выводить на печать в этот же «документ» промежуточные результаты для проверки – и в числовой форме, и в виде графиков.

На скриншоте ниже показаны начальные команды в «документе» с раскодировщиком: это задание пути и имени wav-файла в переменной file1, необязательная команда вывода параметров файла встроенной функцией GETWAVINFO(file1), чтение wav-данных в 2-канальный числовой массив WavData1 встроенной функцией READWAV(file1), вывод количества имеющихся в каналах данных встроенной функцией подсчёта длины массива length(…) :

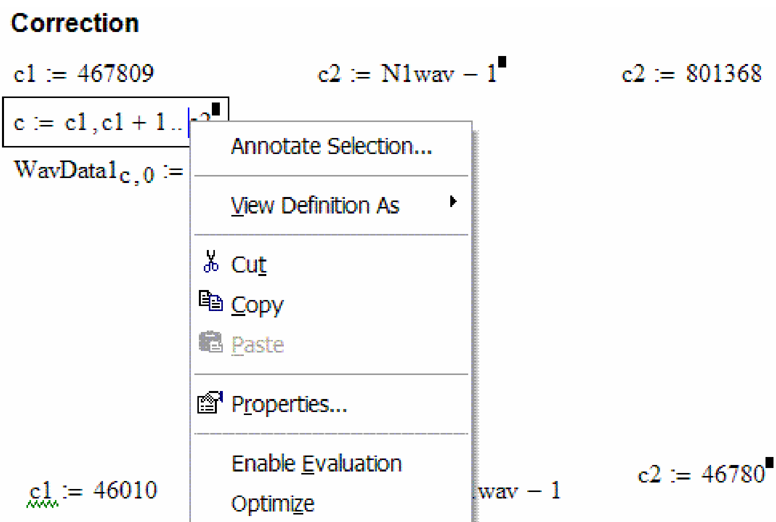


3. Затем в «документе» я поместил область с названием «Correction» – здесь предполагается корректировать числовые данные в массиве WavData1 в случае обнаружения сбойных мест.

Исполнять документ можно много раз (либо частично, командой Calculate Now в меню Tools, – от начала до места, отмеченного красным крестиком, либо полностью, командой Calculate Worksheet, – от начала и до конца), и перед каждым запуском на исполнение можно вносить в документ корректирующие изменения.



Любая процедура может быть выключена или включена командой Disable Evaluation или Enable Evaluation (доступной через меню, открывающееся кликом правой кнопки мыши):



Выключенные процедуры Маткад отмечает значком в виде чёрного квадратика. Сначала, до обнаружения ошибок раскодировки, все процедуры коррекции wav-данных находятся в выключенном состоянии. При обнаружении несовершенств записи можно возвращаться к этой области в документе и включать коррекцию данных, вызывающих ошибку.

Далее для конкретности *<u>рассматриваем раскодировку сигнала с Λ-образными импульсами</u>*.

4. Первая задача в раскодировании – выполнить процедуру, которую я условно называю «нормализацией». Она должна обнаружить пики сигнала в каналах 0 и 1, и сопоставить им нормированные значения – величиной 1 и длительностью в один семпл, а остальным семплам в исходном сигнале будут сопоставлены значения 0. Этот новый 2-канальный массив, состоящий из указанных единиц и нулей, имеет в документе имя Diff2 (так уж сложилось «исторически»); причём, сначала в массив Diff2 переписывается WavData1, подробнее об этом сказано на стр. 6.

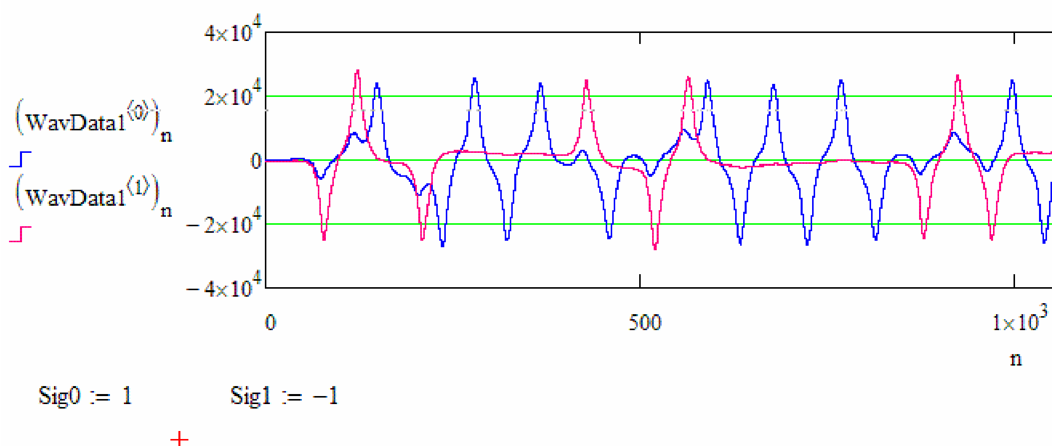
В документе предусмотрены две версии нормализации – «простая» и «улучшенная». Рассмотрим улучшенную версию нормализации. Перед вызовом подпрограммы с этой процедурой надо выполнить ряд вспомогательных действий; а именно:

– просматривая график начала сигнала, определяем полярность самых первых Λ-импульсов в каналах нулей и единиц, и указываем её в переменных с именами Sig0 и Sig1: в случае импульса положительной полярности присваиваем соответствующей переменной значение +1, в случае отрицательной – значение (–1). Пример:

$n1 := 0$

$n2 := n1 + 2000$

$n := n1, n1 + 1 .. n2$

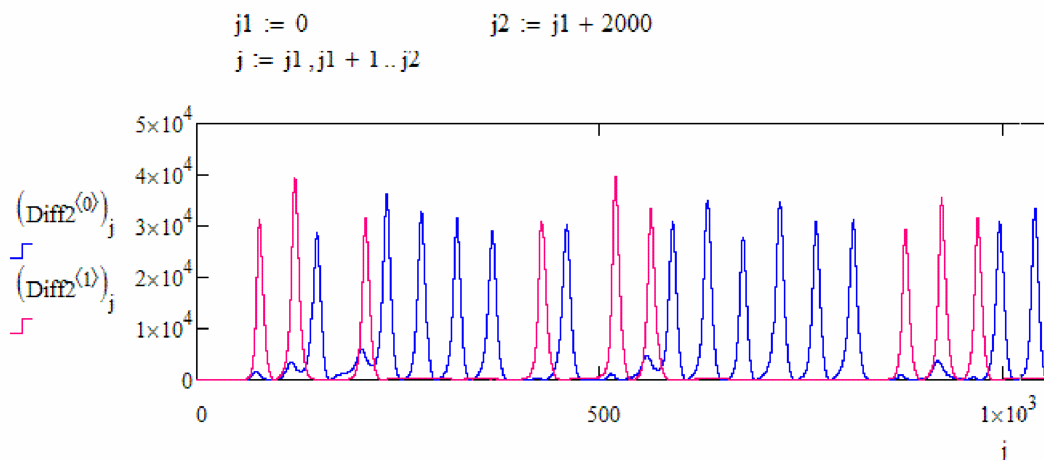


На приведённом здесь рисунке (и в дальнейших рисунках тоже) синим цветом изображается график канала нулей, красным – график канала единиц. Видно, что в канале нулей первый значимый импульс имеет положительную полярность, а в канале единиц – отрицательную. (Видно также, что имеется проникновение сигнала из канала единиц в канал нулей; это уже повод для некоторой коррекции.)

– возводим оба сигнала в квадрат (после того, как они будут преписаны в Diff2) и для удобства делим результат на числовые множители, подбирая их так, чтобы амплитуда получающихся однополярных импульсов была величиной того же порядка, что и амплитуда исходных сигналов, и чтобы она была приблизительно одинаковой в канале нулей и в канале единиц. Т. е. исходная «громкость» каналов в оцифровке не играет большой роли, так как на этом этапе «громкость» однополярных импульсов легко регулируется «программно»:

$$Diff2^{\langle 0\rangle} := \frac{\left(Diff2^{\langle 0\rangle}\right)^2}{20000} \qquad Diff2^{\langle 1\rangle} := \frac{\left(Diff2^{\langle 1\rangle}\right)^2}{20000}$$

Получившиеся однополярные импульсы выводим на график – контролируем их качество:



В этом примере можно заметить ещё одно несовершенство записи (наряду с проникновением канала 1 в канал 0) – импульсы канала 1 расположены во времени несколько позднее, чем следовало бы. В принципе, это не страшно (пока временной сдвиг не настолько велик, что импульсы канала 1 и 0 перекроются). Но лучше всё-таки скорректировать временной сдвиг каналов друг относительно друга, чтобы расположение импульсов 1 и 0 на оси времени выглядело бы как можно более равномерным; тогда возможные «провалы» и прочие сбои раскодировки станут более заметными на графиках, и их будет легче обнаруживать.

Заодно скорректируем «проникновение»: вычитаем из канала 0 сигналы канала 1, умноженные на экспериментально подобранный коэффициент 0.3; обычно это надо делать до корректировки временного сдвига, так как потом, когда исходный сдвиг каналов изменится, вычитание может не дать желаемого эффекта. Итак, возвращаемся в область «Correction» и вставляем там строчку с процедурой вычитания:

$$WavData1^{\langle 0\rangle} := WavData1^{\langle 0\rangle} - 0.3\cdot WavData1^{\langle 1\rangle}$$

Для сдвига мы сначала удлиним массивы семплов в каждом канале на, скажем, 50 семплов, засылая в дополнительные семплы значение «ноль» (опыт показал, что сдвигать импульсы в каком-либо канале более чем на 50 семплов не приходится, удлинение на 50 или 100 семплов является достаточным):

$k := N1wav, N1wav + 1 .. N1wav + 50$ // это цикл по k от N1wav до N1wav + 50

$WavData1_{k,1} := 0 \quad WavData1_{k,0} := 0$ // с шагом 1.

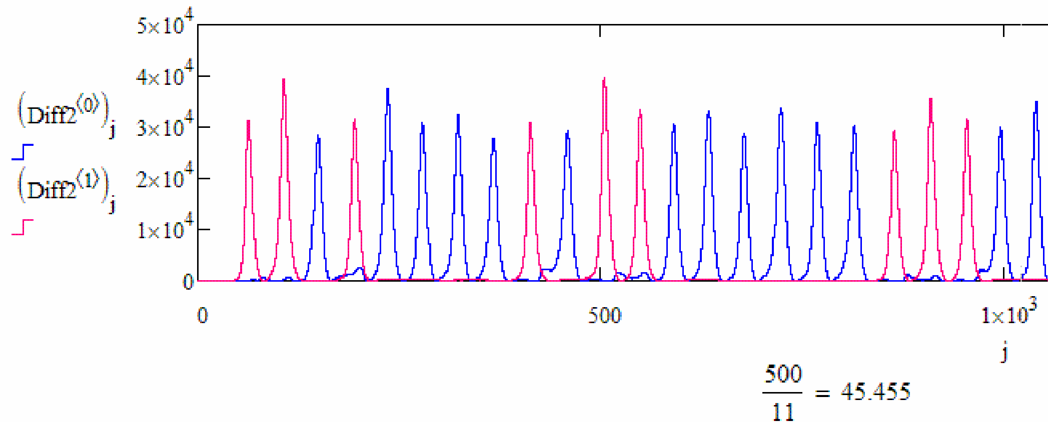
Затем заносим в каналы массива Diff2 сдвинутые каналы WavData1, причём для канала 0 переменная сдвига tau0 равна нулю, а для канала 1 значение переменной сдвига tau, подобранное экспериментально для желаемого сдвига влево, в данном примере равно 15:

$tau := 15 \quad tau0 := 0$

$k := 0, 1 .. N1wav - 1$

$Diff2_{k,0} := WavData1_{k+tau0,0} \qquad Diff2_{k,1} := WavData1_{k+tau,1}$

После этого в массив Diff2 заносятся его значения, возведённые в квадрат и масштабированные, как упоминалось выше. На графике теперь видны однополярные импульсы, получающиеся в результате указанной корректировки:



Из этого же графика видна оценка количества семплов, приходящихся на один импульс: на 500 семплов приходится приблизительно 11 интервалов между импульсами, так что на одном интервале имеем примерно 45 семплов. Исходя из этого, выбираем значение параметра dn, которым в раскодировщике устанавливается минимальный допустимый интервал между нормализованными семплами 1 (значение dn с запасом меньше, чем 45, так как скорость оцифровки может «гулять»). И, глядя на график, задаём значения D0min и D1min – это уровни (пороги) срабатывания «компараторов» в раскодировщике для канала нулей и для канала единиц соответственно (если сигнал превышает порог срабатывания, то нормализованному семплу будет присвоено значение 1, а иначе – значение 0):

$$dn := 35$$

$$D0min := 10000 \qquad D1min := 10000$$

Затем в документе идёт описание (определение) «улучшенной процедуры нормализации» как функции с именем Normalization1impr и с формальными аргументами Diff, S0, S1:

```
Normalization1impr(Diff, S0, S1) := | f0 ← 0
                                    | f1 ← 0
                                    | c0 ← 0
                                    | c1 ← 0
                                    | s0 ← S0
                                    | s1 ← S1
                                    | sig0 ← 1
                                    | sig1 ← 1
                                    | n ← 0
                                    | while n < N1wav
                                    |   | B[n,0] ← 0 if Diff[n,0] < D0min
                                    |   | B[n,0] ← 1 if (Diff[n,0] ≥ D0min) ∧ (f0 = 0)
                                    |   | B[n,0] ← 0 if f0 = 1
```

```
sig0 ← 1  if (WavData1[n+tau0, 0] > 0) ∧ (B[n, 0] = 1)
sig0 ← −1  if (WavData1[n+tau0, 0] < 0) ∧ (B[n, 0] = 1)
B[n, 0] ← 0  if (sig0 ≠ s0) ∧ (B[n, 0] = 1)
s0 ← −s0  if B[n, 0] = 1
f0 ← 1  if B[n, 0] = 1
c0 ← c0 + 1  if (f0 = 1) ∧ (c0 < dn)
f0 ← 0  if c0 ≥ dn
c0 ← 0  if c0 ≥ dn
B[n, 1] ← 0  if Diff[n, 1] < D1min
B[n, 1] ← 1  if (Diff[n, 1] ≥ D1min) ∧ (f1 = 0)
B[n, 1] ← 0  if f1 = 1
sig1 ← 1  if (WavData1[n+tau, 1] > 0) ∧ (B[n, 1] = 1)
sig1 ← −1  if (WavData1[n+tau, 1] < 0) ∧ (B[n, 1] = 1)
B[n, 1] ← 0  if (sig1 ≠ s1) ∧ (B[n, 1] = 1)
s1 ← −s1  if B[n, 1] = 1
f1 ← 1  if B[n, 1] = 1
c1 ← c1 + 1  if (f1 = 1) ∧ (c1 < dn)
f1 ← 0  if c1 ≥ dn
c1 ← 0  if c1 ≥ dn
n ← n + 1
B
```

Понять работу улучшенной процедуры будет легче, если мы сначала разберём всё-таки более простую процедуру «обычной нормализации»:

```
Normalization1(Diff) := | f0 ← 0
                        | f1 ← 0
                        | c0 ← 0
                        | c1 ← 0
                        | n ← 0
                        | while n < N1wav
                        |    B[n, 0] ← 0  if Diff[n, 0] < D0min
                        |    B[n, 0] ← 1  if (Diff[n, 0] ≥ D0min) ∧ (f0 = 0)
                        |    B[n, 0] ← 0  if f0 = 1
                        |    f0 ← 1  if B[n, 0] = 1
                        |    c0 ← c0 + 1  if (f0 = 1) ∧ (c0 < dn)
                        |    f0 ← 0  if c0 ≥ dn
                        |    c0 ← 0  if c0 ≥ dn
```

$$
\left|
\begin{array}{l}
\quad \left| \begin{array}{l}
B_{n,1} \leftarrow 0 \;\; \text{if} \;\; Diff_{n,1} < D1min \\
B_{n,1} \leftarrow 1 \;\; \text{if} \;\; \left(Diff_{n,1} \geq D1min\right) \wedge (f1 = 0) \\
B_{n,1} \leftarrow 0 \;\; \text{if} \;\; f1 = 1 \\
f1 \leftarrow 1 \;\; \text{if} \;\; B_{n,1} = 1 \\
c1 \leftarrow c1 + 1 \;\; \text{if} \;\; (f1 = 1) \wedge (c1 < dn) \\
f1 \leftarrow 0 \;\; \text{if} \;\; c1 \geq dn \\
c1 \leftarrow 0 \;\; \text{if} \;\; c1 \geq dn \\
n \leftarrow n + 1
\end{array} \right. \\
B
\end{array}
\right.
$$

Это – определение пользовательской «функции» (т. е. подпрограммы) с именем Normalization1. Её аргумент здесь обозначен формальным именем Diff. Определение функции предшествует её вызову в «документе»; в дальнейшем, при вызове, на место формального аргумента будет подставлено имя реального аргумента. Последовательность команд в функции пишется «сверху вниз» справа от вертикальной черты. Смысл команд легко понять непосредственно из их написания. В командах можно использовать любые имена переменных, причём новые переменные считаются локальными (т. е. они доступны и действуют только внутри данной функции) и им внутри функции должны быть присвоены значения, а для переменных со старыми именами функция сможет использовать значения, присвоенные им вне функции до её вызова. Присвоение значения внутри функции указывается символом ← .

Если перевести запись команд на язык словесного «псевдокода», то смысл действий в функции Normalization1(Diff) можно пояснить так:

. Сначала присваиваются значения 0 двум флаговым переменным f0 и f1, и двум переменным-счётчикам c0 и c1.
. Затем идёт цикл по переменной n от нуля и до тех пор, пока n остаётся меньшим, чем количество N1wav семплов в каждом из каналов. Содержимое цикла указано справа от ещё одной вертикальной черты. В этом цикле создаётся новый двухканальный массив, с локальным именем B. Его элементы $B_{n,0}$ образуют канал 0, элементы $B_{n,1}$ – канал 1. Присваивание значений элементам массива B происходит следующим образом.
. Если $Diff_{n,0}$ меньше порогового уровня D0min, то засылаем в $B_{n,0}$ ноль; если же $Diff_{n,0}$ равно или больше D0min и флаг f0 равен 0, то засылаем в $B_{n,0}$ единицу, (однако если флаг f0 равен 1, то в $B_{n,0}$ засылаем ноль вне зависимости от $Diff_{n,0}$).
. Затем, если в $B_{n,0}$ была заслана единица, то флагу f0 тоже присваивается значение 1.
. Если флаг f0 равен 1 и значение счётчика c0 меньше, чем dn, то увеличиваем c0 на 1; (тем самым в процессе работы этого цикла реализуется «слепая зона – антидребезг»: после того, как в сигнале канала 0 обнаруживается уровень, равный или превышающий D0min, и устанавливается $B_{n,0} = 1$ (т. е. «срабатывает компаратор»), следующие dn B-семплов в канале 0 будут равны нулю вне зависимости от уровня входного сигнала).
. Если $c0 \geq dn$, то сбрасываем счётчик c0 и флаг f0 в ноль (это конец «слепой зоны»).
. Затем сравниваем $Diff_{n,1}$ с порогом D1min и аналогичным образом формируем $B_{n,1}$.
. В конце тела цикла увеличиваем счётчик цикла n на единицу.
. После выхода из тела цикла в определении функции указано имя массива B; тем самым компилятору Маткада даётся указание: выдать значения массива B в качестве возвращаемых значений функции Normalization1(Diff).

Эта процедура в «документе» вызывается следующим образом:

$$Diff2 := Normalization1(Diff2)$$

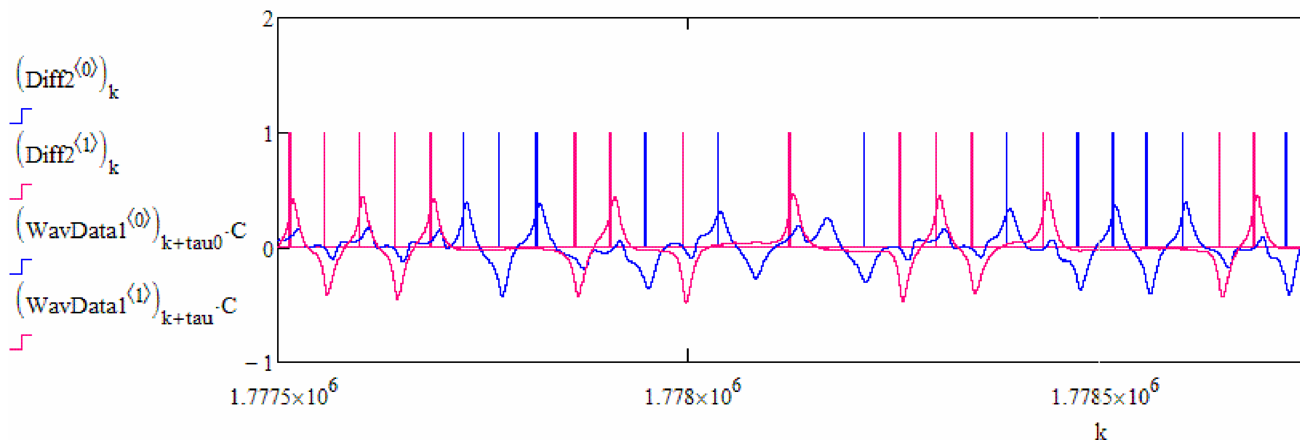
В роли входного двухканального массива здесь выступают однополярные импульсы Diff2, а возвращаемые нормализованные семплы заносятся для экономии памяти в массив с тем же именем Diff2.

Работает эта функция, в общем-то, неплохо, но не идеально. Основной её недостаток поясняется следующим примером. Контроль (о нём речь пойдёт ниже) показывает, что при раскодировке wav-файла, указанного в начале, происходит сбой в районе 1800000-го семпла. Просмотр графиков Diff2 совместно с WavData1 в этом районе обнаруживает небольшой «провал» сигнала в канале 0 – из-за этого провала теряются два нормализованных семпла величиной 1 в канале нулей. (Подчеркну важное для нас свойство Маткада: допускается в любом месте документа вставить команду построения графиков, причём на одном рисунке можно изобразить несколько графиков и выбрать для них различные цвета, а интервал с участком для просмотра легко регулируется в широких пределах. В примере ниже интервал просмотра задаётся переменной k, которая с шагом 1 пробегает значения от k1 до k2.)

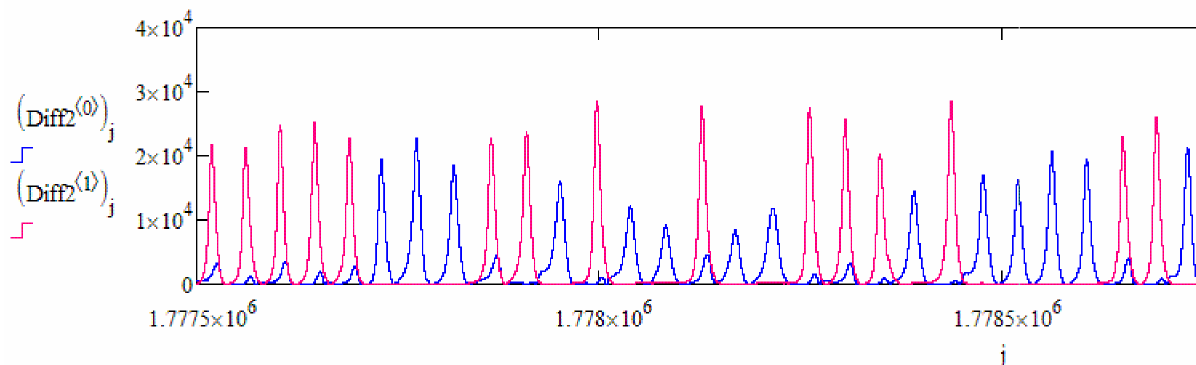
$k1 := 1777500$ $\quad C = 0.00002 \quad D0min = 10000 \quad D1min = 10000$

$k2 := k1 + 2000$

$k := k1, k1 + 1 .. k2$



Вернувшись назад, к просмотру графиков однополярных импульсов, видим, что «порог компаратора» D0min желательно понизить до значения, наверное, порядка 6000:

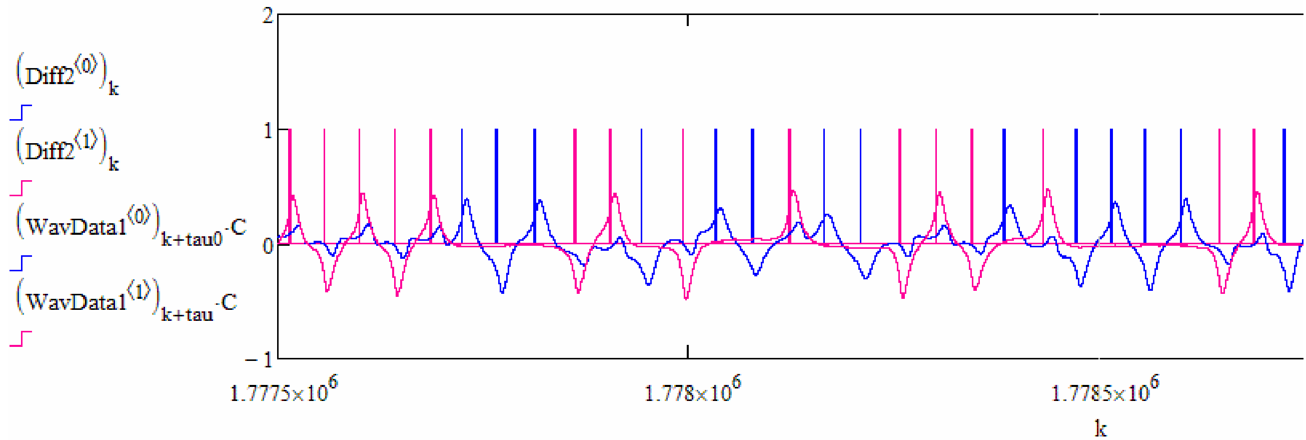


И действительно, с новым значением, D0min = 6000, нет прежней потери нормализованных сэмплов:

k1 := 1777500 C = 0.00002 D0min = 6000 D1min = 10000

k2 := k1 + 2000

k := k1 ,k1 + 1 .. k2

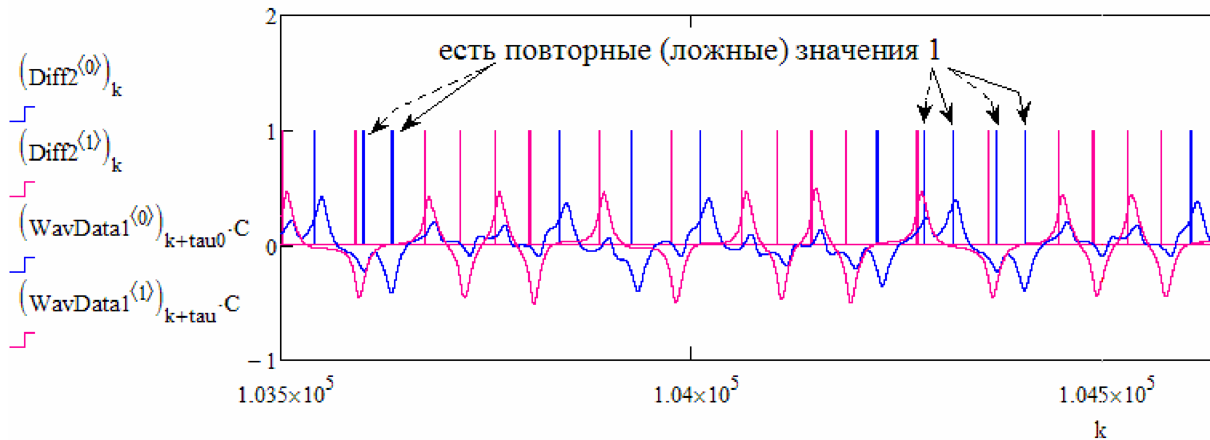


Однако теперь контроль показывает, что сбои всё равно есть, причём первый сбой раскодировки происходит значительно раньше, – там, где его не было, – в районе 100000-го семпла. Просмотром графиков Diff2 совместно с WavData1 в этом районе мы обнаруживаем причину сбоя:

k1 := 103500 C = 0.00002 D0min = 6000 D1min = 10000

k2 := k1 + 2000

k := k1 ,k1 + 1 .. k2



Видно, что сигнал канала 0 в этом участке в данном примере имеет довольно большую амплитуду и большое «проникновение» канала 1 в канал 0, поэтому при пониженном пороге компаратора возникают повторные (ложные) его срабатывания – появляются лишние значения 1 в массиве нормализованных семплов канала нулей $Diff2^{<0>}$.

«Улучшенная» функция Normalization1impr(Diff,S0,S1) создаёт массив нормализованных семплов без повторных (лишних) значений 1. С целью не допущения лишних единиц в ней используются вспомогательные локальные переменные: для канала 0 это переменная sig0 с текущим знаком wav-сигнала и переменная s0 с ожидаемым знаком wav-сигнала для нормализованного семпла 1. Для канала 1 аналогичные переменные это sig1 и s1. Идея основана на том, что соседние Λ-пики wav-сигнала в одном и том же канале, которым функция

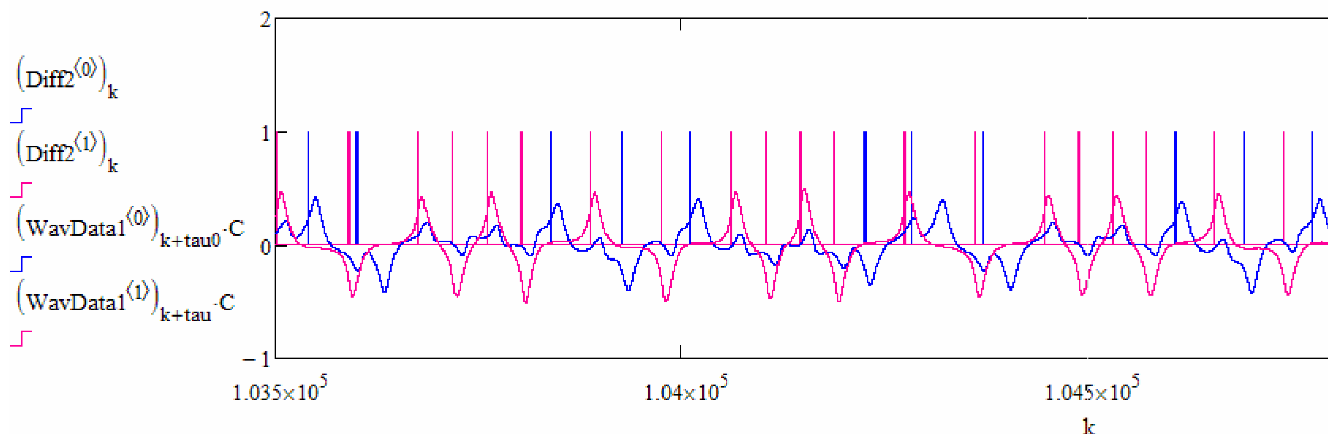
должна сопоставить нормализованные семплы 1, как видно из графиков сигнала, обязательно имеют противоположную полярность (тогда как лишний семпл 1 появляется при значении wav-сигнала с той же самой полярностью, что и у предшествующего семпла 1 в том же канале). Весь «псевдокод» функции Normalization1impr(Diff,S0,S1) здесь не выписываю, он в основном такой же, как и у простой функции Normalization1(Diff); отмечу только строки, в которых учитывается указанная корреляция полярностей соседних истинных Λ-пиков wav-сигнала:

. Начальные ожидаемые значения s0 и s1 берутся из входных аргументов S0 и S1.
. После того, как произошло обнаружение пика в массиве однополярных импульсов канала нулей и произведена предварительная засылка 1 в $B_{n,0}$, проверяется, что полярность соответствующего n-го семпла в массиве WavData1 соответствует s0; если соответствия нет, то $B_{n,0}$ сбрасывается в 0 (т. е. эта «единица» считается ложной).
. Если же соответствие есть (т. е. «единица» оказалась верной), то изменяем знак в s0, тем самым подготавливаемся к обнаружению пика противоположной полярности.
. Аналогичные действия выполняются и при формировании массива $B_{n,1}$.

При работе с «улучшенной» процедурой простую функцию Normalization1 мы дезактивируем (выключаем), вместо неё вызывается «улучшенная» функция:

Diff2 : = Normalization1impr(Diff2, Sig0, Sig1)

Прежнее построение графиков при запуске «документа» с улучшенной процедурой нормализации показывает, что повторные (лишние) семплы 1, появлявшиеся в канале 0, теперь отсутствуют. Правда, нужные семплы 1 в данном примере кое-где по-прежнему привязаны к неправильным пикам сигнала в канале нулей – возникшим из-за проникновения помехи из канала единиц. Из-за этого расположение нормализованных семплов 1 в каналах нулей и единиц выглядит неравномерным, но очерёдность их следования здесь не нарушена, так что этот конкретный участок больше не должен быть источником ошибок раскодировки:



К сожалению, улучшенная процедура не решает всех проблем; как мы увидим ниже, контроль обнаруживает присутствие ошибок раскодировки в других участках этого примера.

Продолжаю, однако, рассказывать о содержании «документа» по порядку.

5. Следующая задача после нормализации массива семплов – получить из этого массива последовательность нулей и единиц, которую затем можно будет разбить на последовательность байт-кодов. Это делает функция BitStream(Diff), определённая следующим образом:

```
d := 1
BitStream(Diff) := | err ← 0
                   | j ← 0
                   | b ← 0
                   | n ← 0
                   | while  n < (N1wav − 2)
                   |    | a0 ← Diff[n,0]
                   |    | a1 ← Diff[n,1]
                   |    | c ← 1  if  (a0 > 0) ∨ (a1 > 0)
                   |    | c ← 0  if  (a0 = 0) ∧ (a1 = 0)
                   |    | err ← err + 1  if  (a0 > 0) ∧ (a1 > 0)
                   |    | b ← 0  if  b = 1
                   |    | b ← 1  if  (b = 0) ∧ (c > 0)
                   |    | B[j] ← 0  if  (b = 1) ∧ (a0 = 1)
                   |    | B[j] ← 1  if  (b = 1) ∧ (a1 = 1)
                   |    | j ← j + 1  if  (b = 1)
                   |    | n ← n + d  if  (b = 1)
                   |    | n ← n + 1  if  b ≠ 1
                   | B[j] ← err
                   | B
```

Вот, что должно происходить в этой подпрограмме:

. Инициализируются значением 0 переменные: err – счётчик ошибок типа «наложение семплов 1 в каналах 0 и 1 друг на друга» (вероятность такой ошибки мала, но предусмотреть её надо, так как эту ошибку очень трудно заметить на графике), n – счётчик для цикла по сэмплам входного массива, j – счётчик для бит в выходном массиве, b – вспомогательный флаг.
. В цикле по n выполняются следующие действия:
  . в переменные a0 и a1 заносим нормированные семплы каналов 0 и 1 соответственно;
  . полагаем c = 1, если a0 или a1 не равно нулю, а иначе полагаем c = 0;
  . увеличиваем err на единицу, если a0 и a1 одновременно отличны от нуля;
  . сбрасываем b в ноль, если было b = 1;
  . устанавливаем b в единицу, если b = 0 и c = 1;
  . полагаем $B_j = 0$, если b = 1 и a0 = 1;
  . полагаем $B_j = 1$, если b = 1 и a1 = 1;
  . увеличиваем j на единицу, если b = 1;
  . увеличиваем n на d, если b = 1, а иначе увеличиваем n на единицу.
. После выхода из цикла по n в последний элемент массива B заносится число err.
. Этот массив B позиционируется в конце как «возвращаемое значение функции».

Параметр d определяет «слепую зону»: если обнаружен информационный бит, то анализ семплов продолжается с пропуском следующих d семплов. (Тем самым предусматривается возможность исключить слишком близко расположенные семплы «единица», рассматривая их как ошибку. В данном примере из-за сильной неравномерности положений семплов такая «слепая зона» не используется: значение d выбрано равным всего лишь 1.)

При вызове этой функции аргументом является массив нормализованных семплов Diff2, а возвращаемый массив бит получает имя Bits1:

$$Bits1 := BitStream(Diff2)$$

$$Nbits1 := length(Bits1)$$

$$Nbits1 = 5.1578 \times 10^4$$

$$Bits1_{Nbits1-1} = 0$$

$$\frac{Nbits1 - 1}{9} = 5730.777777777777$$

$$Num2 := trunc\left(\frac{Nbits1 - 1}{9}\right)$$

$$Num2 = 5730$$

Bits1 =

	0
0	1
1	1
2	0
3	1
4	0
5	0
6	0
7	0
8	1
9	0
10	1
11	1
12	0
13	0

Nbits1 это длина получившегося массива Bits1. Так как нумерация элементов массивов начинается с нуля, то последний элемент (содержащий число err) имеет номер $Nbits1 - 1$. Вывод его значения в документе показывает, что в данном примере err = 0, т. е. ошибок типа «наложения семплов» нет. На скриншоте виден также вывод массива Bits1 в вертикальную таблицу (столбец) с полосой прокрутки; в общем-то такая таблица нам не нужна, она осталась в документе «исторически» – на стадии отладки процедур она позволяла убедиться в соответствии раскодированных первых бит картине Λ-пиков на графиках сигналов.

Важным (но тоже не достаточным) индикатором правильности раскодировки является проверка целочисленности значения $(Nbits1 - 1) / 9$. Количество раскодированных бит в массиве Bits1 равно $Nbits1 - 1$; единицу здесь надо вычитать из количества элементов Nbits1 потому, что последний элемент это не бит, а число err. Поскольку за каждым байтом, т. е. за каждыми восемью битами должен следовать контрольный бит чётности, то $(Nbits1 - 1) / 9$ должно равняться количеству байт, которое при правильной раскодировке должно получиться целым. Видно, что в нашем примере оно не целое, т. е. на данном этапе раскодировка пока ещё идёт с ошибками. Встроенная функция Маткада trunc(…) возвращает целую часть числа. Целая часть от $(Nbits1 - 1) / 9$ обозначена в документе как Num2, она нужна для продолжения процесса раскодировки вне зависимости от того, есть ошибки или нет.

Кстати, если на кассете или в сопроводительных бумагах к оцифровке указано количество байт (что, к сожалению, бывает не часто), то из сравнения его с Num2 можно оценить количество ошибок в текущем массиве Bits1 и тем самым хоть как-то оценить объём

дальнейшей работы по их исправлению. Обычно Num2 оказывается меньше истинного числа байт из-за пропусков нормализованных семплов вследствие «провалов» амплитуды в оцифрованном сигнале. Если Num2 больше истинного количества байт, то это говорит нам о том, что порог срабатывания компаратора в одном или в обоих каналах выбран слишком низким, так что в массиве нормализованных семплов появляются лишние единички. Если же Num2 совпадает с истинным количеством байт, то этим ещё не доказывается правильность раскодировки, так как при нестабильной амплитуде сигнала в оцифровке у нас в разных участках раскодировки могут быть одновременно и пропуски семплов и лишние семплы.

6. Следующая задача в процессе раскодировки – получить из массива бит Bits1 массив байт-кодов. Но сначала проверяем чётность предполагаемых байт путём сравнения с «девятыми битами чётности», и затем удаляем эти девятые биты; это делает функция Bits2(Bits):

```
Bits2(Bits) := | error ← 0
               | n_e ← 0
               | for k ∈ 0,1..Num2 − 1
               |    | S ← 0
               |    | for j ∈ 0,1..7
               |    |    | B[j+k·8] ← Bits[j+k·9]
               |    |    | S ← S + B[j+k·8]
               |    | p ← S − 2·trunc(S/2)
               |    | p ← p − Bits[k·9+8]
               |    | error ← error + 1 if p ≠ 0
               |    | n_e ← k if error = 1
               | B[Num2·8] ← error
               | B[Num2·8+1] ← n_e
               | B
```

В этой подпрограмме переменная error подсчитывает количество ошибок чётности, а в переменную n_e заносится номер того байта, в котором обнаруживается самая первая ошибка чётности. В конце анализа значения error и n_e присваиваются соответственно предпоследнему и последнему элементам возвращаемого массива. Вызов этой подпрограммы выглядит так:

BitsOut : = Bits2(Bits1)

Полученный таким образом массив BitsOut содержит Num2·8 нулей и единиц без «девятых битов чётности». Он может быть разбит на байты, если исключить из него два последних элемента, представляющих собой не биты, а числа error и n_e. Однако верить таким байтам можно только в том случае, когда числа error и n_e равны нулю; в нашем примере это пока ещё не так:

$$BitsOut_{Num2\cdot 8} = 465$$
$$BitsOut_{Num2\cdot 8+1} = 564$$

В данном примере первая ошибка чётности обнаружилась в 564-ом байте.

Для того чтобы установить причину ошибки и выбрать способ её исправления, надо найти неверный нормализованный семпл в массиве нормализованных семплов Diff2. Этой цели служит функция SampleNum(N, Diff). Она у нас принимает два аргумента: номер бита, в котором обнаружилась ошибка чётности (это номер байта, в котором обнаружилась ошибка чётности, умноженный на 9), и имя массива нормализованных семплов. Возвращаемым значением является номер семпла, в районе которого возникает ошибка:

```
SampleNum(N, Diff) := | n ← 0
                      | nb ← 0
                      | while  nb < N
                      |     | nb ← nb + 1  if  Diff[n,0] = 1
                      |     | nb ← nb + 1  if  Diff[n,1] = 1
                      |     | n ← n + 1
                      | n
```

Вызов этой функции и вывод номера семпла, в районе которого возникает ошибка, в документе выглядит так:

$$BadSample := SampleNum\left(BitsOut_{Num2 \cdot 8+1} \cdot 9, Diff2\right)$$

$$BadSample = 217768$$

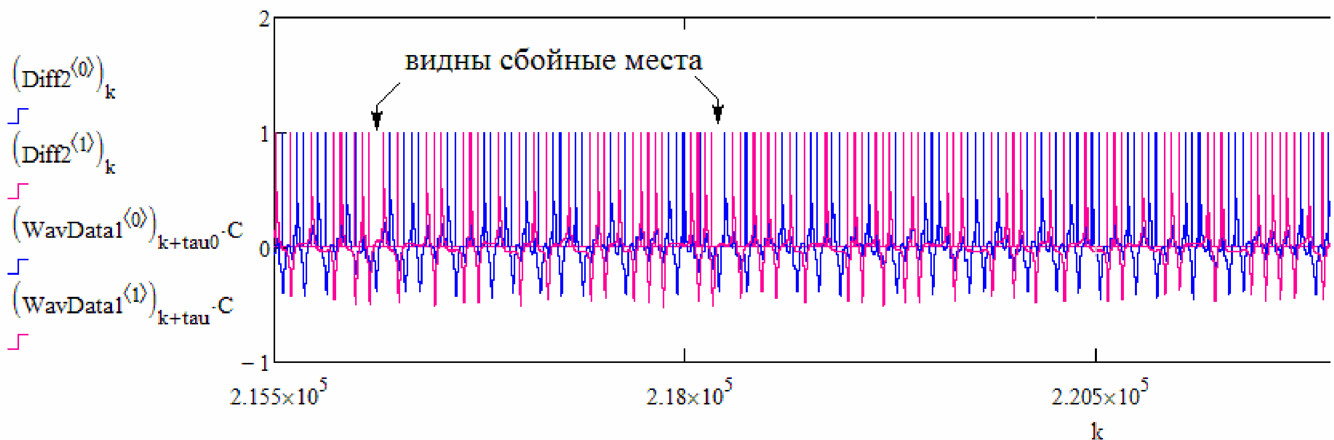
Опыт показывает, что на самом деле ошибка раскодировки может возникать раньше, так что проявляться в виде неверной чётности она будет лишь через несколько байт (тоже неправильных) или даже десятков байт, т. е. – спустя сотни и даже тысячи семплов от сбойного места. Поэтому номер «плохого семпла» BadSample не следует воспринимать как точное значение; он лишь ориентировочно указывает район для поиска ошибки.

Возвращаемся в документе назад – к просмотру графиков нормализованных семплов совместно с оцифровкой сигнала. Для расширения зоны обзора увеличиваем разницу между начальным и конечным номером просматриваемых семплов с 2000 до 10000:

$k1 := 215540$ $\qquad C = 0.00002 \qquad D0min = 6000 \qquad D1min = 10000$

$k2 := k1 + 10000$

$k := k1, k1 + 1 .. k2$

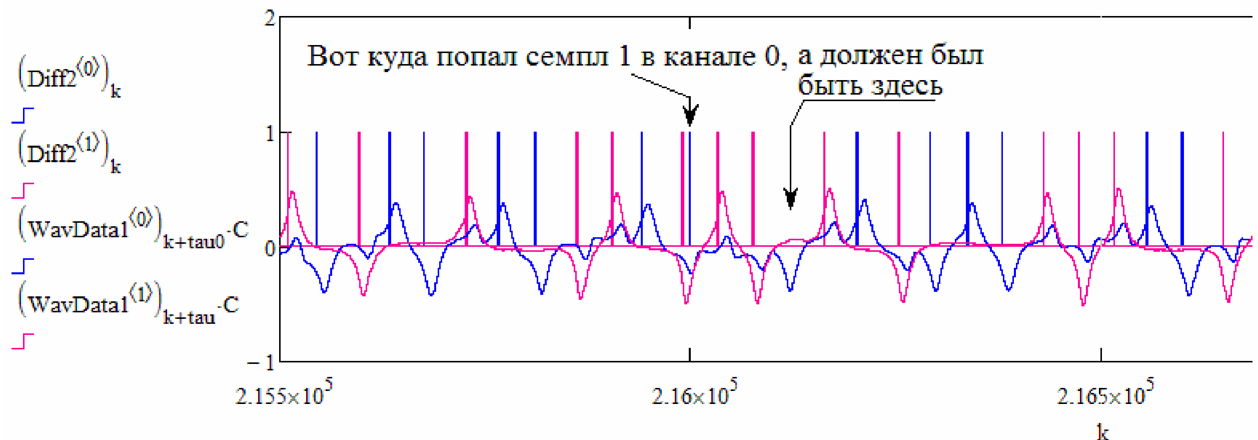


«Поймав» таким образом первое сбойное место, снова ограничиваем интервал обзора двумя тысячами семплов, чтобы лучше разглядеть форму оцифрованного сигнала в сбойном месте и понять причину сбоя:

$k1 := 215540$ $\quad C = 0.00002 \quad D0min = 6000 \quad D1min = 10000$

$k2 := k1 + 2000$

$k := k1, k1 + 1 .. k2$



Видно, что в данном примере сбой произошёл из-за проникновения канала 1 в канал 0. Хотя мы и пытались скомпенсировать такое проникновение, глядя на картину начала файла, но, по-видимому, величина проникновения в разных участках разная; здесь она довольно большая, и компаратор канала 0 среагировал на ложный пик.

Есть два способа исправлять подобные ошибки (по трудоёмкости они примерно равны):
1) изменять значения элементов в массиве нормализованных семплов Diff2, т. е. находить элемент массива Diff2 с ошибочным семплом 1, приравнивать его нулю, а элементу в том месте, где должен быть семпл 1, присваивать значение 1.
2) в начале документа – в области с названием «Correction» – изменять значения элементов массива WavData1, корректируя форму сигнала в сбойном месте.

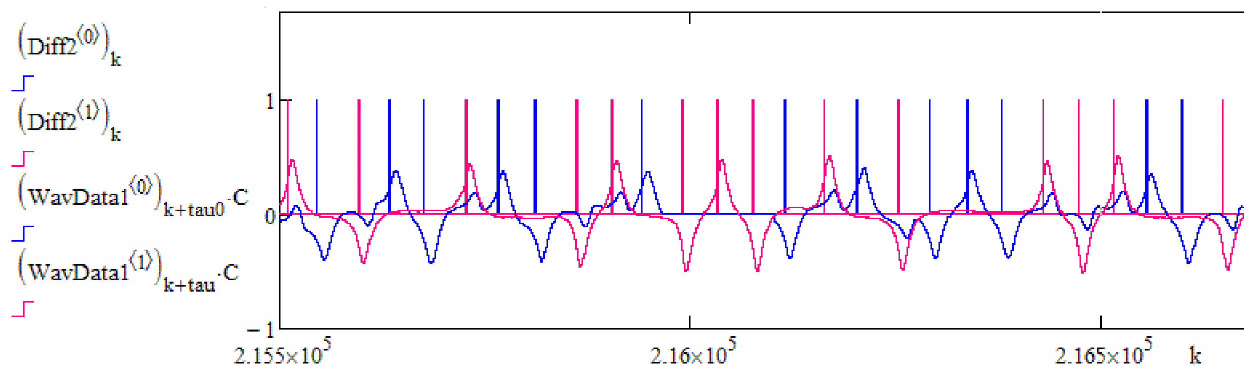
Применим второй способ. Отыскав с помощью графика номера элементов $WavData1^{<0>}$, соответствующие ложным пикам, присвоим этим элементам значение 0 (цикл с таким присваиванием нулей помещаем после той строчки, где задаётся корректирующее вычитание канала 1 из канала 0):

$c1 := 216005 \qquad c2 := c1 + 135$

$c := c1, c1 + 1 .. c2$

$WavData1_{c,0} := 0$

Повторное исполнение документа показывает, что картина исправилась желаемым образом:



Аналогичным образом исправляем и замеченное раньше второе сбойное место, в районе 218000-го нормализованного семпла. После этих исправлений очередная первая ошибка чётности обнаруживается уже в 1664-ом байте; ему соответствует $BadSample = 644886$. Исправляем и её, но обнаруживаются ещё ошибки. И так далее, и так далее, и так далее…

Поскольку в данном примере обнаруживается очень много ошибок указанного выше типа, то для борьбы сразу со всей их совокупностью пробуем подобрать более точно параметр D0min. Раньше мы уменьшили его значение от 10000 до 6000. Теперь же увеличим этот порог, чтобы компаратор канала 0 перестал «цепляться» за каждый маленький ложный пичок. При $D0min = 7000$ первая ошибка чётности отодвигается уже в район 4000-х байтов, т. е. ближе к концу файла. По ходу дела выявились также провалы амплитуды в канале 1; для устранения соответствующих ошибок чётности оказалось достаточно снизить D1min от 10000 до 8000. Ещё одна неприятность в данном примере: начиная примерно с 1890000-х семплов возросло проникновение канала 1 в канал 0, в результате чего в канале 0 снова пошли ложные срабатывания компаратора. Для борьбы с их совокупностью вставляем в области «Correction» (ниже уже имеющихся там строк коррекции) цикл дополнительной компенсации проникновения канала 1 в канал 0:

$$c1 := 1890000 \qquad c2 := 1970000$$
$$c := c1, c1 + 1 .. c2$$
$$WavData1_{c,0} := WavData1_{c,0} - 0.3 \cdot WavData1_{c,1}$$

Аналогично – циклами в определённых участках оцифровки – можно корректировать и другие несовершенства сигнала: изменять амплитуду импульсов в том или ином канале, не только подавлять ложные пики, но и выращивать недостающий пик в нужном месте. (В примерах встречаются разнообразные дефекты сигнала; «проникновение» каналов – это частный случай.)

Однако, если оцифровка большая и корректирующих поправок сделано много, то может негативным образом проявиться склонность Маткада к точным вычислениям и к услужливости пользователю. Все числа представляются в Маткаде в формате double – с плавающей запятой с «двойной» точностью. В 32-разрядной операционной системе каждое такое число занимает 8 байт памяти. Для удобства пользователя программа Маткад, пока она не закрыта, сохраняет в памяти все версии числовых данных, чтобы пользователь мог через меню Edit выполнять многократные “undo” – отмены поправок, не затрачивая много времени на повторные расчёты. В итоге, после очередной коррекции массивов, ОС может выдать нам, как это ни смешно, сообщение о нехватке памяти:

$$Diff2^{\langle 0 \rangle} := \frac{\left(Diff2^{\langle 0 \rangle}\right)^2}{20000} \qquad Diff2^{\langle 1 \rangle} := \frac{\left(Diff2^{\langle 1 \rangle}\right)^2}{20000}$$

Not enough memory for this operation.

В этом случае приходится сохранить (командой Save в меню File) текущий текст документа и закрыть программу Маткад, чтобы ОС освободила занимаемые Маткадом ресурсы. После этого можно снова открыть документ и продолжить работу с ним. Чтобы избежать скорого повторения нехватки памяти следует ниже всех строк с коррекцией вставить и выполнить команду записи скорректированных данных в новый, отредактированный wav-файл

```
file2 := "c:\L_Work\test-progs\for_d3-28\k67a_t444-465_red-1.wav"
WRITEWAV(file2 , 44100 , 16) := WavData1
```

после чего дезактивировать (выключить) эту команду и все строки с коррекцией. В дальнейшей работе надо будет загружать в документ отредактированный таким образом файл.

И вот, наконец, в результате хлопот с коррекциями в нашем примере появляется целочисленное количество байт в раскодировке: $(Nbits1 - 1) / 9 = 5731$. И при этом счётчики ошибок выдают долгожданные нули:

$$BitsOut_{Num2 \cdot 8} = 0$$

$$BitsOut_{Num2 \cdot 8+1} = 0$$

Разбивать массив бит на байты и формировать из них массив с количеством элементов $Num2 = 5731$ будет функция Bytes1(Bits). Байты в ней вычисляются в тетрадно-десятичной форме:

```
Bytes1(Bits) := | for k ∈ 0, 1 .. Num2 − 1
                |   | m ← k·8
                |   | Byte_k ← Bits_{m+7} + 2·Bits_{m+6}
                |   | Byte_k ← Byte_k + 4·Bits_{m+5} + 8·Bits_{m+4}
                |   | Byte_k ← Byte_k + 100·(Bits_{m+3} + 2·Bits_{m+2})
                |   | Byte_k ← Byte_k + 100·(Bits_{m+1}·4 + 8·Bits_m)
                | Byte
```

Вот вызов этой функции и просмотр результата в вертикальной таблице с прокруткой:

$BytesOut := Bytes1(BitsOut)$

BytesOut =

	0
0	1300
1	600
2	1403
3	2
4	11
5	1301
6	0
7	413
8	608
9	407
10	1514
11	1300
12	0
13	413

Один из признаков успешной раскодировки – код 0512 в последнем байте:

5727	600
5728	413
5729	8
5730	512

7. Теперь можно подсчитать контрольную сумму. Для этого можно было бы определить специальную (пользовательскую) функцию, но в данной версии документа контрольная сумма вычисляется сермяжно, в строках. Контрольная сумма обозначена как КР, тетрады n-го байта обозначены как B_n и A_n , вычисление тетрад ведётся в цикле по n, а их суммирование по номерам байтов (от нулевого до предпоследнего) задаётся стандартным математическим символом суммы:

$$n := 0, 1 .. Num2 - 1$$

$$B_n := \mathrm{trunc}\left(\frac{BytesOut_n}{100}\right)$$

$$A_n := BytesOut_n - B_n \cdot 100$$

$$N1 := 0$$

$$N2 := Num2 - 2$$

$$N2 = 5729$$

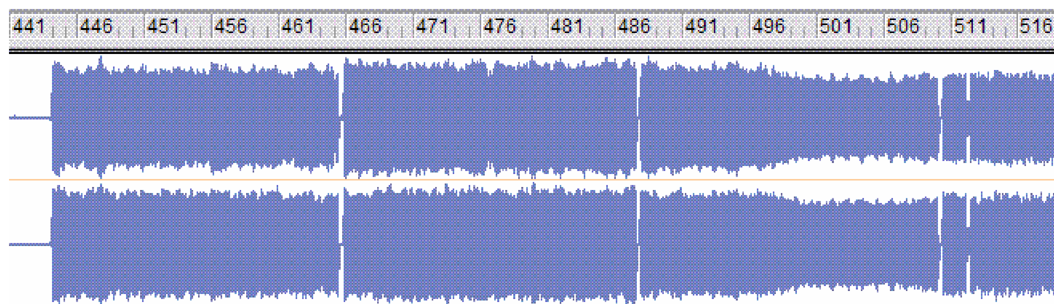
$$KP := \sum_{k = N1}^{N2} \left(B_k + A_k\right)$$

$$KP = 73716$$

8. Итоговое действие – запись полученных байт-кодов из массива BytesOut в txt-файл:

```
file3 := "c:\L_Work\test-progs\for_d3-28\k67a_t444-465_KP-73716_N-5730.txt"
WRITEPRN(file3) := BytesOut
```

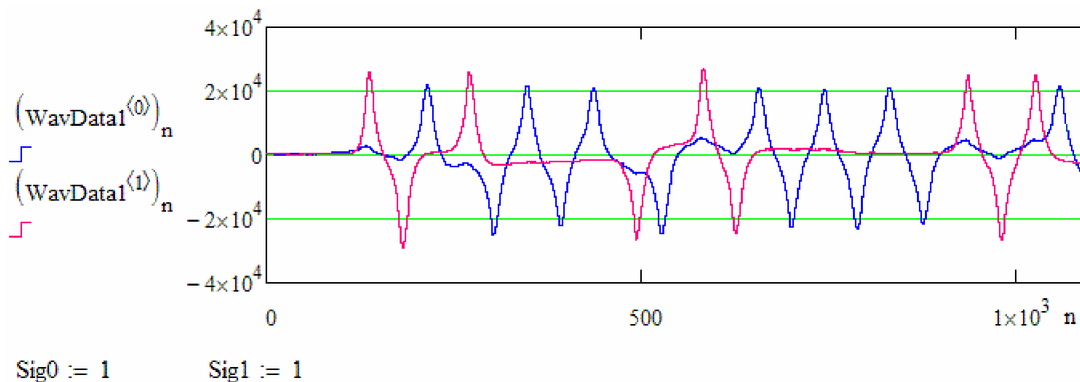
Ну и, казалось бы, всё? Можно радоваться? Увы, нет. Поскольку ошибок по ходу этой раскодировки было очень много, то верить первому образовавшемуся значению КР нельзя. Для проверки желательно раскодировать ещё одну подобную запись. Поэтому возвращаемся в аудио-редактор и смотрим, нет ли в оцифровке чего-нибудь похожего на копию раскодированного wav-файла:



Раскодировали эту запись,
а за ней есть вроде бы копия,
попробуем и её раскодировать

9. Затем действуем так, как уже было рассказано в разделе **1** (см. стр. 1) и далее. Нужную копию записываем в отдельный wav-файл. В нашем примере это будет файл k67a_t465-487.wav, числа 465-487 в его имени – временн*ы*е метки, указывающие положение записи в оцифровке (файл с предыдущей записью назывался k67a_t444-465.wav, а после редактирования – k67a_t444-465_red-1.wav). Для работы с ним записываем копию предыдущего «маткадного документа», чтобы не создавать документ с нуля, а воспользоваться готовыми настройками и значениями параметров в процедурах (однако все строки коррекции выключаем).

Загружаем в этот новый документ файл k67a_t465-487.wav. Определяем полярность самых первых Λ-импульсов в каналах 0 и 1, и указываем её в переменных Sig0 и Sig1:

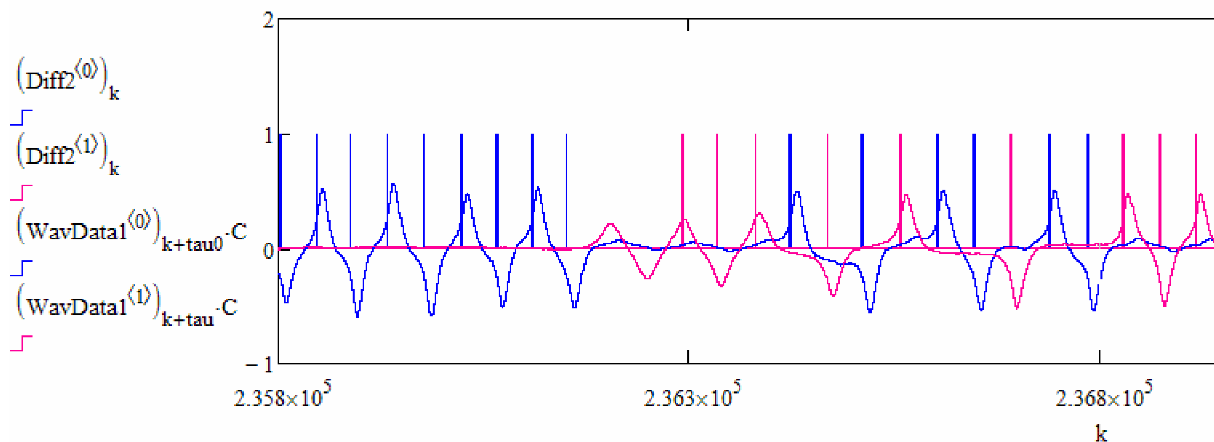


Запускаем документ на исполнение. Первая ошибка чётности обнаруживается в байте с номером 600, ей сответствует BadSample = 238800. Визуальным поиском ошибки в этом районе на графиках семплов выявляется причина ошибки – «провал амплитуды» в канале 1:

k1 := 235800　　C = 0.00002　　D0min = 7000　　D1min = 8000

k2 := k1 + 2000

k := k1 ,k1 + 1 .. k2

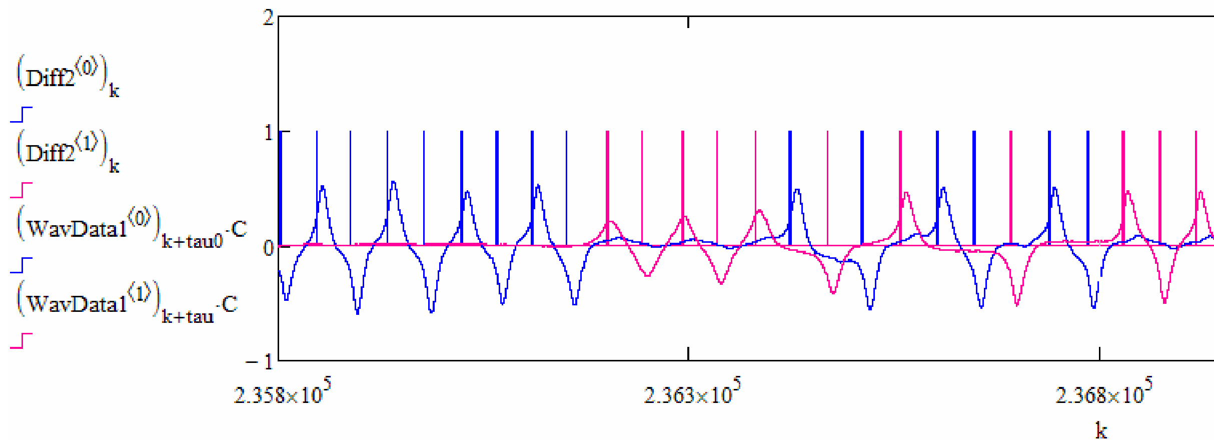


Видно, что в данном файле импульсы по своей форме выглядят лучше, чем в предыдущем файле: проникновение канала в канал почти отсутствует. В таких случаях наиболее типичными несовершенствами оцифровки являются «провалы амплитуды». При этом (т. е. при хорошей форме импульсов) целесообразно попробовать не исправлять каждый отдельно взятый «провал» в канале 1, а понижением порога D1min нейтрализовать сразу всю их совокупность (хотя мы на этом этапе ещё не знаем, много ли в файле имеется провалов, и не узнаем, если они не проявятся в контроле). В данном примере так и получилось при выборе D1min = 5000 :

$k1 := 235800$ $\qquad C = 0.00002$ $\qquad D0min = 7000$ $\qquad D1min = 5000$

$k2 := k1 + 2000$

$k := k1, k1 + 1 .. k2$



В данном примере после указанного изменения D1min не потребовалось прибегать ни к каким другим исправлениям: количество байт в раскодировке получилось целочисленным, (Nbits1 – 1) / 9 = 5731, счётчики ошибок чётности выдали нули:

$$BitsOut_{Num2 \cdot 8} = 0$$

$$BitsOut_{Num2 \cdot 8+1} = 0$$

Подсчёт контрольной суммы даёт число 73735 (как оказывается, присутствующее в описании кассеты, и это хорошо!):

$$KP = 73735$$

Записываем получившуюся раскодировку в новый txt-файл:

```
file3 := "c:\L_Work\test-progs\for_d3-28\k67a_t465-487_KP-73735_N-5730.txt"
WRITEPRN(file3) := BytesOut
```

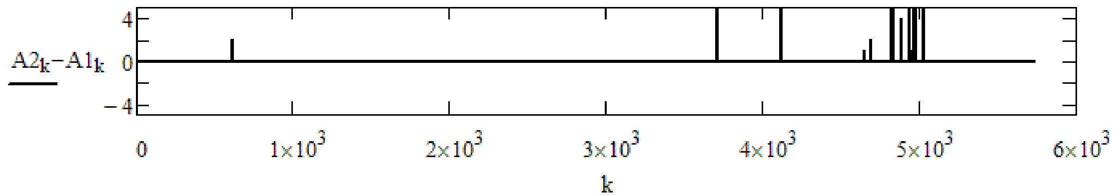
10. Итак, *предположительно* для одной и той же *(?)* программы мы имеем две раскодировки без признаков ошибок, но с двумя разными контрольными суммами. Что делать дальше?

Резонно предполагать, что более надёжной является та раскодировка, которая получена с меньшим количеством исправлений, притом из оцифровки с хорошей формой сигналов, да ещё и, вроде бы, с верной контрольной суммой. Тогда план дальнейших действий может быть таким: путём побайтного сравнения обоих txt-файлов мы найдём номера различающихся байтов, по ним определим районы поиска сбойных семплов, и будем пытаться искать сбои и исправлять их в первую очередь в «подозрительной» раскодировке (в нашем примере это первая раскодировка), а если там не найдём, тогда уж – в более «надёжной» (т. е. во второй).

Побайтное сравнение раскодировок в Маткаде делаем в отдельном документе. Для этого читаем байт-коды раскодировок в массивы с именами A1 и A2 :

```
file1 := "c:\L_Work\test-progs\for_d3-28\k67a_t444-465_KP-73716_N-5730.txt"
file2 := "c:\L_Work\test-progs\for_d3-28\k67a_t465-487_KP-73735_N-5730.txt"
A1 := READPRN(file1)          A2 := READPRN(file2)
```

И смотрим обзорный график разности этих массивов; на нём видны (в форме вертикальных «палок») отличные от нуля разности байт-кодов:



Затем находим номера этих не совпадающих байтов. Вот описание функции BadBytes, которая находит такие номера, строчка с её вызовом, и вывод списка номеров в таблицу:

```
BadBytes(M1,M2) := | N ← length(M1)
                   | j ← 0
                   | for n ∈ 0,1..N − 1
                   |    | Bj ← n  if  M1n − M2n ≠ 0
                   |    | j ← j + 1  if  M1n − M2n ≠ 0
                   | B
```

List := BadBytes(A1,A2)

List =

	0
0	615
1	3712
2	3714
3	4120
4	4123
5	4650
6	4691
7	4826
8	4829
9	4889
10	4939
11	4947
12	4963
13	4979
14	5028
15	

(Кстати, с помощью символа T из панели инструментов Vector and Matrix Toolbar в Маткаде вертикальный «вектор-столбец» можно легко транспонировать в горизонтальную таблицу

$List^T$ =

	0	1	2	3	4	5	6	7	8	9	10
0	615	3712	3714	4120	4123	4650	4691	4826	4829	4889	4939

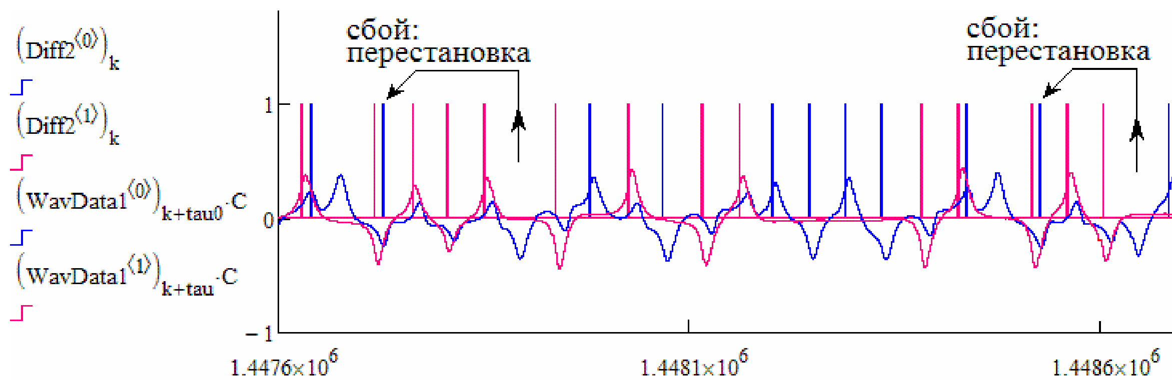
с полосой прокрутки.)

Теперь открываем документ с «подозрительной» раскодировкой и пытаемся искать подозрительные семплы (в нашем примере это был первый документ, в него загружается файл k67a_t444-465_red-1.wav). Раньше мы уже оценили количество семплов в одном бите – примерно 45. Значит, на один байт вместе с битом чётности приходится около 45·9 семплов. Берём из списка “List” номер первого подозрительного байта (615), умножаем его на 45·9, и получаем тем самым оценку для номера подозрительного семпла:

$$45 \cdot 9 \cdot 615 = 249075$$

В этом районе просматриваем графики семплов. И… да, видим не замеченный ранее сбой: между 237500-ым и 238000-ым семплами поменялись местами единички в каналах 1 и 0 из-за ложного пика в канале 0. Когда подобные перестановки бит происходят внутри байтов, они не выявляются проверкой чётности; заметить такие сбои крайне трудно – их можно обнаружить только путём тщательного визуального обследования картины семплов...

Устранив этот сбой (дополнительным корректирующим вычитанием канала 1 из канала 0 в районе семплов 230000 – 250000), переходим к визуальному поиску следующей возможной ошибки – в районе 45·9·3712 = 1503360-го семпла. И там обнаруживаем череду подобных сбоев:



Пробуем устранить сразу всю их совокупность корректирующим вычитанием (до 1890000-го семпла, потому что после этого семпла вычитание уже делалось раньше, см. стр. 18):

$$c1 := 1400000 \qquad c2 := 1890000$$

$$c := c1, c1 + 1 .. c2$$

$$WavData1_{c,0} := WavData1_{c,0} - 0.2 \cdot WavData1_{c,1}$$

Результат: видимые сбои исчезают, индикаторы ошибок по-прежнему показывают 0, количество байт – целое, как и было: 5731. Однако контрольная сумма, хотя она и менялась по ходу исправлений, случайно оказывается прежней: КР = 73716.

Тем самым мы имеем здесь реальный поучительный пример: у разных раскодировок с одинаковым количеством байт, без признаков ошибок на индикаторах чётности, вполне могут совпасть контрольные суммы; причём – с неправильны значением, как выясняется дальше. Различие раскодировок подтверждается повторным вычислением списка номеров байт, отличных от байт в якобы хорошем файле с КР = 73735; до внесения исправлений различались 15 байт, а после – уже только 10 (т. е. можем надеяться, что 5 ошибок в «подозрительной» раскодировке уже удалось исправить):

$List^T =$

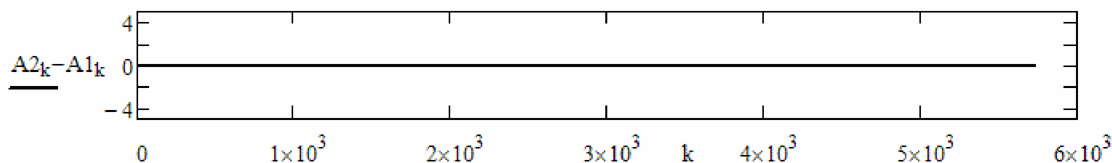
	0	1	2	3	4	5	6	7	8	9	10
0	4046	4048	4049	4829	4889	4939	4947	4963	4979	5028	

Дальнейший поиск ошибок показывает, что в предыдущем корректирующем вычитании получился «перелёт» (да, и такое бывает): вычитать надо было примерно до 1570000-го семпла; затем оставить как есть, а с 1602000-го по 1700000-й опять вычитать, и так далее. Т. е., к сожалению, «одним махом» устранить все ошибки не удаётся; это типичная ситуация – трудоёмкий поиск и локальное исправление ошибок в раскодировке, штука за штукой.

В данном конкретном примере этот процесс завершился так: все ошибочные байты обнаружились именно в «подозрительном» файле, и после их исправления воспроизвелась та же контрольная сумма КР = 73735, что и у раскодировки «якобы хорошего» файла (который, таким образом, вероятнее всего действительно является хорошим). Подвергшийся коррекции wav-файл сохраняем в итоге с именем k67a_t444-465_red-3_KP-73735_N-5730.wav, а его раскодировку – с именем k67a_t444-465_KP-73735_N-5730.txt. Побайтное сравнение этой раскодировки с раскодировкой хорошего файла выглядит так:

```
file1 := "c:\L_Work\test-progs\for_d3-28\k67a_t444-465_KP-73735_N-5730.txt"   // читаем байт-коды
file2 := "c:\L_Work\test-progs\for_d3-28\k67a_t465-487_KP-73735_N-5730.txt"   // раскодировок в массивы
A1 := READPRN(file1)          A2 := READPRN(file2)                           // с именами A1 и A2
```

На обзорном графике разности этих массивов не видно отличных от нуля разностей байт:



Более строгой проверкой одинаковости обеих раскодировок является вычисление суммы S абсолютных величин побайтных разностей:

$$N1 := \mathrm{length}(A1) \qquad N1 = 5731$$

$$N2 := \mathrm{length}(A2) \qquad N2 = 5731$$

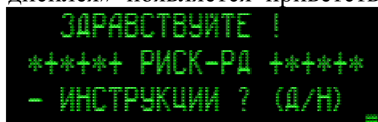
$$N := N1$$

$$S := \sum_{n=0}^{N-1} \left| A1_n - A2_n \right| \qquad S = 0$$

Результат S = 0 означает, что обе раскодировки совпадают полностью.

Дополнительным индикатором правильности раскодировки может служить также её запуск в имитаторе Д3-28: успешный запуск и работа программы в имитаторе подтверждают верность раскодировки (но неудача не доказывает ошибочность раскодировки, так как в самом имитаторе, возможно, есть ошибки, а также могут присутствовать какие-то неучтённые особенности запуска программы).

В рассмотренном конкретном примере с программой k67a_t444-465_KP-73735_N-5730.txt особенность запуска есть: эту программу надо загружать на начальный адрес 24576, а перед запуском надо байт 0206 на шаге 24576+409 = 24985 заменить байтом 0514. Тогда после запуска (клавишей S) и включения режима «ДУП-ЛИН» (клавишей F5) на экране «фрязинского дисплея» появляется приветствие:

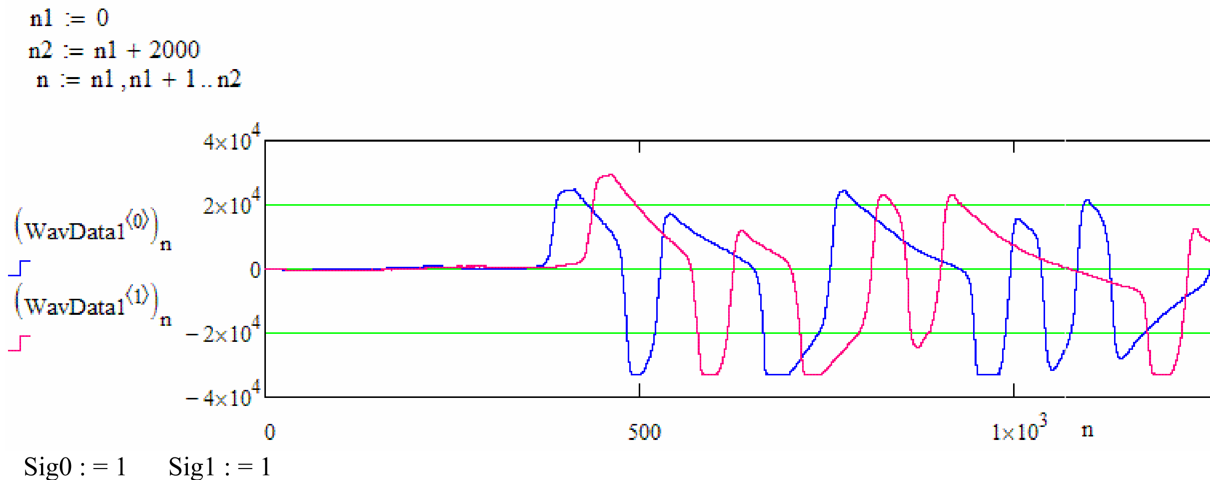


Это одна из начальных версий редактора документов Виталия Колесника, на стадии её разработки.

11. Вкратце рассмотрим *шаги в раскодировке wav-записей сигналов с «П-образными» импульсами*. Для примера возьмём файл tape_frmj__KP-38246.wav. Начинается «документ» с раскодировщиком так же, как в рассмотренном выше случае. Сначала загружаем wav-файл:

```
file1 := "c:\L_Work\test-progs\for_d3-28\tape_frmj__KP-38246.wav"
WavData1 := READWAV(file1)
N1wav := length(WavData1<0>)
                                    N1wav = 1.433939 × 10^6
```

В области «Коррекция» всё выключаем или удаляем. Для улучшенной процедуры нормализации указываем полярность Sig0 и Sig1 первых импульсов в канале 0 и в канале 1, определяя эту полярность по графику сигналов в начале файла:



Удлиняем 2-канальный массив семплов, чтобы иметь возможность сдвигать каналы во времени:

```
k := N1wav , N1wav + 1 .. N1wav + 50                  // это цикл по k от N1wav до N1wav + 50
WavData1[k,1] := 0        WavData1[k,0] := 0           // с шагом 1.
```

Задаём величины сдвигов (в данном примере они выбраны равными нулю в обоих каналах; ненулевой сдвиг потребовался бы для отстающего канала, но в данном примере никакой относительной задержки каналов не выявляется):

$$tau0 := 0 \quad tau := 0$$

Задаём экспериментально подобранный «шаг дифференцирования» dk, и в цикле по k от 0 до dk с шагом 1 засылаем нули в начальные элементы двухканального массива Diff2:

$$dk := 9$$
$$k := 0 \,..\, dk$$
$$Diff2_{k,0} := 0 \qquad Diff2_{k,1} := 0$$

Заносим в Diff2 результат «дифференцирования» импульсов WavData1:

$$k := dk, dk+1 \,..\, N1wav - 1$$
$$Diff2_{k,0} := WavData1_{k+tau0,0} - WavData1_{k+tau0-dk,0}$$
$$Diff2_{k,1} := WavData1_{k+tau,1} - WavData1_{k+tau-dk,1}$$

В этих трёх строчках и заключается отличие раскодировки П-образных импульсов от Λ-образных. (При раскодировке Λ-образных импульсов три указанные выше строчки должны быть дезактивировны (выключены); вместо них включаются три другие строчки, с помощью которых в Diff2 переписываются каналы WavData1:

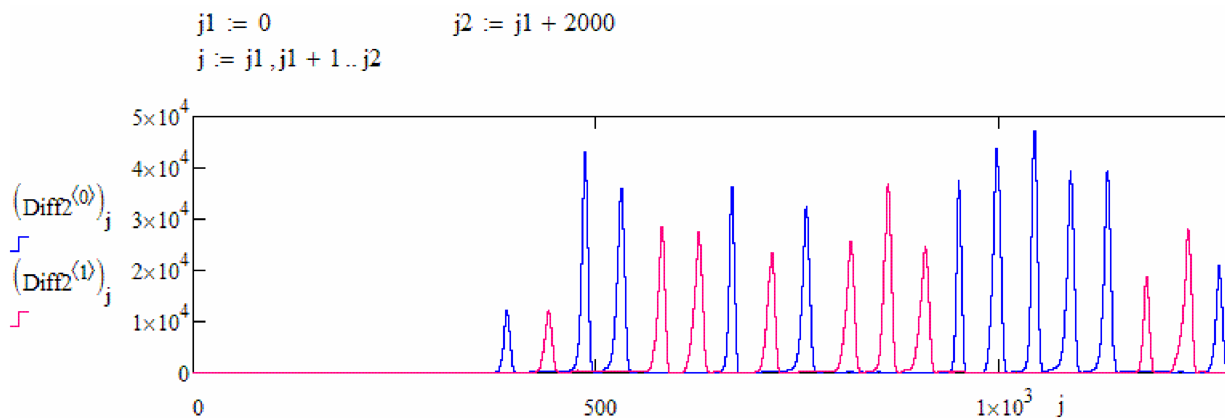
$$k := 0, 1 .. N1wav - 1$$
$$Diff2_{k,0} := WavData1_{k+tau0,0}$$
$$Diff2_{k,1} := WavData1_{k+tau,1}$$

Сейчас же, т. е. при раскодировке П-образных импульсов эти три строчки выключены.)

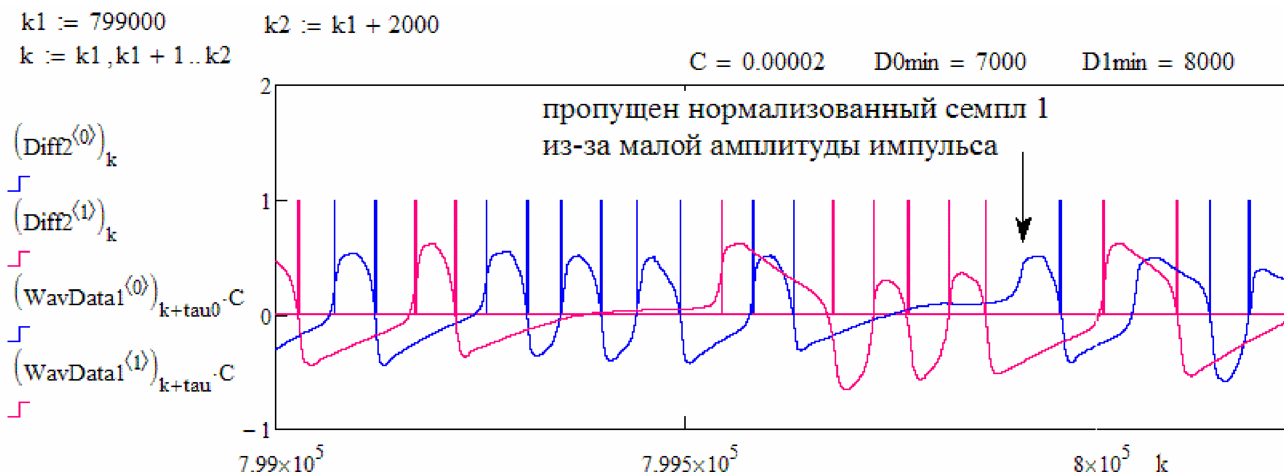
Дальнейшие строки в документе такие же, как и при раскодировке Λ-образных импульсов. Возводим результат дифференцирования во вторую степень, чтобы получить однополярные импульсы:

$$Diff2^{\langle 0 \rangle} := \frac{\left(Diff2^{\langle 0 \rangle}\right)^2}{20000} \qquad Diff2^{\langle 1 \rangle} := \frac{\left(Diff2^{\langle 1 \rangle}\right)^2}{20000}$$

Выбрать подходящие пороги срабатывания «компараторов» помогают графики получившихся однополярных импульсов:



Если взять значения порогов из предыдущего примера – D0min : = 7000, D1min : = 8000 для функции Normalization1impr(Diff2, Sig0, Sig1), – то обнаруживается ошибка:



Выбором D0min : = 5000 эта ошибка устраняется, и оказывается, что другие ошибки отсутствуют: файл без проблем раскодируется до конца, с правильной контрольной суммой KP = 38246.

Больше ничего отличительного в раскодировке оцифровок с П-образными импульсами нет; отличие от случая Λ-импульсов есть только в наблюдаемой на графиках форме сигналов и в необходимости указанного выше «дифференцирования» П-импульсов.

12. Для проверки правильности раскодировки программы огромную (можно сказать – основную) роль играет знание верной контрольной суммы и количества байт. Хорошо, когда такая информация имеется в сопроводительной документации к программе; но где её искать, если желаемой документации нет?

Оказывается, в «фортрано-подобных» записях желаемая информация содержится в заголовочных файликах, которые присутствуют в виде трёх копий перед тремя копиями самой программы. (Между тремя заголовочными файлами и файлами программы есть ещё буферный файл длиной 82 байта, см. рис. в конце страницы 2. Если раскодировка буферного файла идёт плохо, то можно вместо него взять любые 80 байт, отличные от 0512, и добавить в конце 0515 0512; при нумерации байтов с нуля код 0512 должен быть на шаге 81. Буферный файл обязан иметь указанную длину (и код 0512) в конце, а его содержимое не играет роли при работе Д3-28 с «фортрано-подобными» записями: главное – заголовок и сама программа.) Рассмотрим этот сюжет немного подробнее.

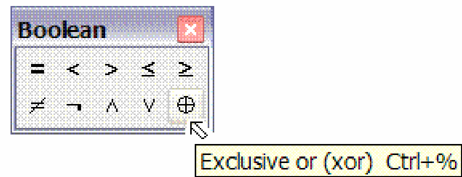
Понятно, что раскодировать все три копии заголовка нет необходимости; достаточно одной. Для примера, допустим, мы выделили из оцифровки «Кассеты 58 В» один из заголовков в отдельный wav-файлик k58b_h1_t63.wav. Раскодировка этого заголовка с использованием «улучшенной нормализации» при D0min : = 7000 и D1min : = 6000 проходит без проблем (файл-то короткий!). Вот его содержимое с описанием назначения отдельных байт-кодов:

BytesOut =

	0	
0	5	– два каких-то служебных байта; возможно, они означают тип программы.
1	107	
2	1409	– 4 символа имени программы. Если из старшей тетрады вычитать 8, то получаются байт-коды 0609 0614 0613 0612: это буквы ИНМЛ
3	1414	
4	1413	
5	1412	
6	9	– это номер байта перед END в программе: 9·256 + 15·16 + 5·1 = 2549
7	1505	
8	0	– номер записи (равен трём) в библиотеке программ на данной МЛ.
9	3	
10	302	– в этих трёх байтах (0302 0002 0300) тетрады изображают прямо десятичные цифры контрольной суммы программы: КР = 32023
11	2	
12	300	
13	0	
14	0	
15	0	
16	0	
17	5	– здесь всегда пятёрка; не знаю, что она означает.
18	1414	– контрольный байт заголовка для проверки командой VEX R_i R_j
19	0	
20	512	– код 0512 обязательной команды END в конце файла.

Всё это выяснилось опытным путём. В частности, выяснилось, что контрольный байт заголовочного файлика (байт с номером 18 при отсчёте с нуля) вычисляется по байтам с номерами 0, 1, … , 17 посредством команды VEX – это сумма по всем указанным байтам одноимённых бит в байтах по модулю 2 (суммирование операцией XOR, исключающим «или»).

Хотя в коротких заголовочных файликах незамеченные ошибки раскодировки маловероятны, всё же лучше не лениться и выполнять подсчёт контрольного байта. В Маткаде панель «логических инструментов» позволяет выбрать для суммирования нулей или единиц операцию XOR, она обозначена там как «плюс в круге»:



Вот описание нашей пользовательской функции, вычисляющей контрольный байт:

```
Vex(Bytes, n1, n2) := | z ← 0
                      | for n ∈ n1, n1 + 1 .. n2
                      |   | B ← trunc(Bytes_n / 100)
                      |   | A ← Bytes_n − B·100
                      |   | D_(n−n1) ← B·16 + A
                      | N ← length(D)
                      | for k ∈ 0, 1 .. 7
                      |   S_k ← 0
                      | for m ∈ 0, 1 .. N − 1
                      |   | v ← 0
                      |   | for k ∈ 7, 6 .. 0
                      |   |   | b_k ← trunc((D_m − v) / 2^k)
                      |   |   | v ← v + b_k·2^k
                      |   |   | S_k ← S_k ⊕ b_k
                      | Dec ← S_0
                      | for k ∈ 1, 2 .. 7
                      |   Dec ← Dec + S_k·2^k
                      | B ← trunc(Dec / 16)
                      | A ← Dec − B·16
                      | Out ← B·100 + A
                      | Out
```

Если бы в Маткаде не было встроенной операции XOR, то вместо неё мы применили бы простенькую самодельную функцию:

$$\mathrm{OurXOR}(a,b) := \begin{array}{|l} c \leftarrow 1 \\ c \leftarrow 0 \ \text{if} \ a = b \\ c \end{array}$$

При этом в описании функции $\mathrm{Vex}(\mathrm{Bytes}, n1, n2)$ достаточно было бы заменить строчку $S_k \leftarrow S_k \oplus b_k$ строчкой $S_k \leftarrow \mathrm{OurXOR}(S_k, b_k)$.

Аргументами нашей функции $\mathrm{Vex}(\mathrm{Bytes}, n1, n2)$ являются массив байтов и номера начального и конечного байта в этом массиве, по которым вычисляется XOR-сумма. Вызов этой функции и сравнение результата с 18-м байтом раскодировки показывают, что заголовочный файл раскодирован верно:

$$\mathrm{VexKS} := \mathrm{Vex}(\mathrm{BytesOut}, 0, 17)$$

$$\mathrm{VexKS} = 1414 \qquad \mathrm{BytesOut}_{18} = 1414$$

Информация о контрольной сумме и о количестве байт «фортрано-подобной» программы, полученная из заголовочного файла, облегчает её раскодировку (раскодировать достаточно, конечно, только одну из трёх копий программы). Если в нашем примере выделить из оцифровки «Кассеты 58 В» в отдельный wav-файл (k58b_p3_t65-75.wav) первую из трёх копий программы, записанную на МЛ после трёх копий заголовка и буферного файла, то она без проблем раскодируется «улучшенным» способом при D0min : = 3000 и D1min : = 5000. В раскодировке программы можно увидеть не только 4 символа имени, но символы из его расширения:

$\mathrm{BytesOut}^T =$

	0	1	2	3	4	5	6	7	8	9	10
0	5	200	1409	1414	1413	1412	215	302	303	13	...

Здесь первые два байта, по-видимому, являются какими-то служебными. Затем идут те же четыре буквы, что и указанные в заголовочном файле, а затем – расширение имени (байты 0215 0302 0303) в виде символов /23, так что полное имя этой программы есть ИНМЛ/23. Такое же имя присутствует в описании кассеты 58 В, и это хорошо.

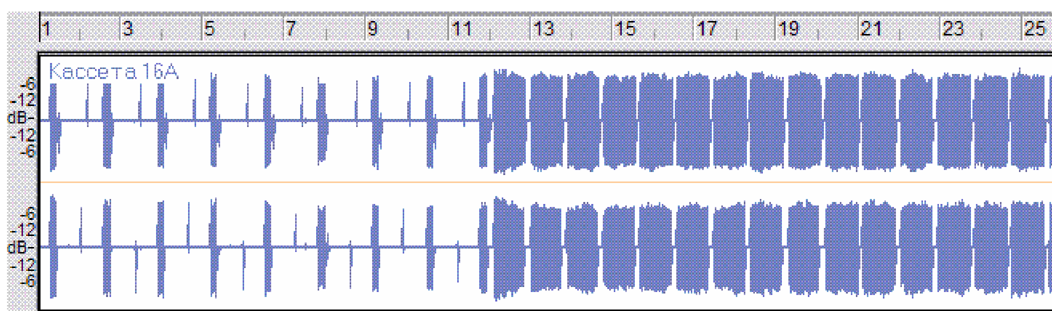
Для чтения «фортрано-подобных» раскодировок в имитаторе следует составлять из них txt-файл с таким же количеством копий и с такой же очерёдностью заголовочных, буферных и программных файлов, как и на МЛ (за образец можно взять готовые примеры). Файлы отделяются друг от друга строками с пробелами. В конце такого txt-файла надо три раза (тоже через строки с пробелами) записать пару байт 0000 0512; это имитация трёхкратной записи «КБ», означающей «конец библиотеки». Она необходима для успешного просмотра каталога МЛ командами Фортрана или ОС ВТ-МХТИ. Командами этих систем производится и чтение раскодировок в имитаторе Д3-28.

Запись текста пользовательской программы в системе «Выстра» тоже имеет составную структуру, но без дублей. Сначала в ней идёт короткий заголовок с именем программы (после букв имени в нём есть только байт 0010 – по-видимому, это код команды «перевод строки», и 0512 – код команды END). Затем идёт второй заголовочный файлик; в нём первый байт-код это 0000, а после него идут байты, изображающие цифры контрольной суммы программы. Например, контрольная сумма 53809 представляется байт-кодами 0505 0303 0808 0000 0909. И затем идёт основной файл с текстом программы на языке «Выстра». Для чтения в имитаторе Д3-28 (конечно, под управлением «Выстры») надо из раскодировок всех трёх файлов составить txt-файл с такой же трёхфайловой структурой; файлы в ней должны отделяться друг от друга строками с пробелами.

Раскодировка 256-байтных блоков, записанных командой SAVE R

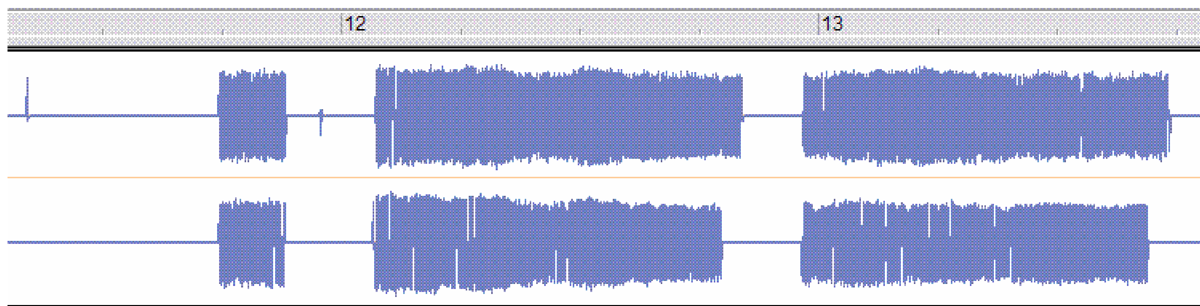
13. В общих чертах раскодировка 256-байтных блоков подобна рассмотренной выше раскодировке «моноблока» произвольной длины: она состоит из аналогичных этапов – а) осмотр графиков сигнала в оцифровке и «нарезка» оцифровки на wav-файлы с одиночными блоками для их поштучного раскодирования; б) собственно применение к каждому блоку процедур раскодирования; в) самое главное: «танцы с бубном» для проверки правильности результата, устранение обнаруженных ошибок, сборка txt-файла. Однако имеется определённая специфика; рассмотрим её на примере оцифровки «Кассета 16 А» с программой ВТ-9Р.

а) Осмотр начала оцифровки в аудио-редакторе показывает, что там присутствуют 9 очень коротких файлов – это 9 копий загрузчика, – а затем идут 256-байтные блоки:



Загрузчик раскодируется обычным образом (описанным выше) без проблем, так как он короткий; в данном примере – всего 40 байт (код END на 39-м шаге при отсчёте байтов с нуля, контрольная сумма КР = 610). Однако, как показывает дизассемблирование этого загрузчика, он потому такой короткий, что загружает он не систему, а лишь второго загрузчика, который будет загружать систему ВТ-МХТИ 9Р. Таким образом, здесь проявился существенный момент: чтобы разобраться с картиной 256-байтных блоков, необходимо вникать в дизассемблированный текст 1-го загрузчика. В частности, как оказывается, 1-й загрузчик засылает в регистр R_{11} два байта 0015 0404, которые играют роль контрольной суммы для 2-го загрузчика, записанного в формате 256-байтного блока.

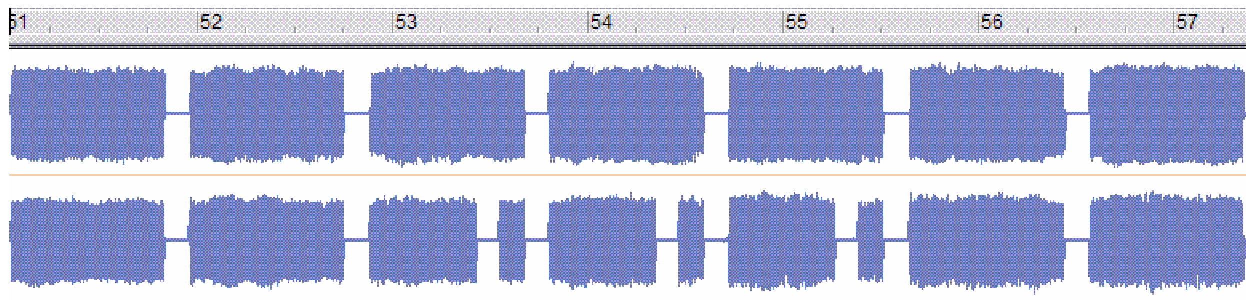
Рассматривая картину нескольких блоков, следующих за 1-м загрузчиком, в растянутом по времени масштабе, замечаем в данном примере их приметную особенность – в каждом блоке канал нулей выглядит более длинным, чем канал единиц (очевидно, из-за присутствия в конце таких блоков длинной серии бит 0):



Значит, в этом примере нам повезло: по указанному приметному признаку мы можем определить количество копий 2-го загрузчика, не прибегая к раскодировке всех 256-байтных

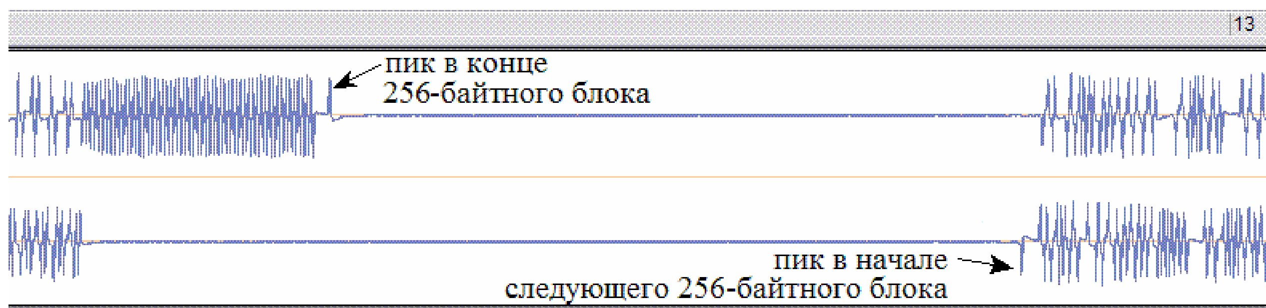
блоков подряд. Конкретно, в этом примере обнаружилось 9 блоков 2-го загрузчика. Вслед за ними идут 162 блока, затем – коротенькая пауза со «щелчком», а после неё – ещё 9 + 162 = 171 штук 256-байтных блоков, причём 9 блоков там имеют тот же самый приметный признак (т. е. они являются копиями 2-го загрузчика).

Таким образом, в данной оцифровке система ВТ-9Р записана два раза. На одну запись приходятся 162 блока без учёта загрузчиков, но мы пока ещё не знаем, все ли эти блоки различны. Зато мы уже знаем, как обнаруживать одинаковые блоки, – по какой-нибудь приметной особенности в изображении их графиков. Просматривая с этой целью в аудио-редакторе всё изображение оцифровки, обнаруживаем, в частности, вот такую картину, указывающую на то, что в данном примере блоки записаны с 3х-кратным копированием:



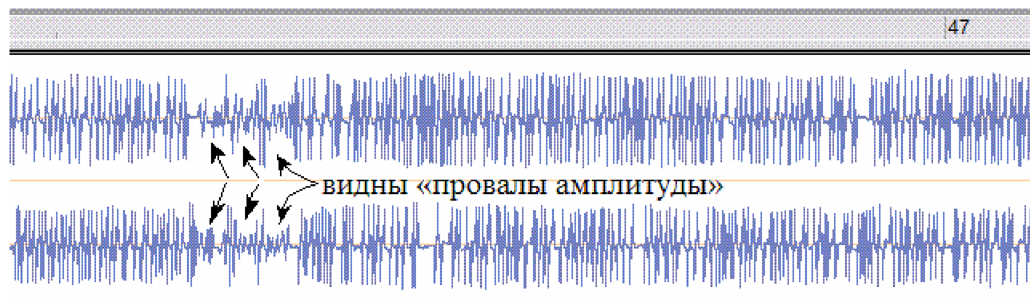
Следовательно, в данной оцифровке имеется 162 / 3 = 54 основных блока, подлежащих раскодировке; плюс 2-й загрузчик. Каждый из этих блоков следует записать в отдельный wav-файл.

При такой записи участков из оцифровки следует учитывать специфическое свойство формы 256-байтных блоков – в конце и в начале блока имеется «пик» («щелчок» сигнала):



Эти «пики» не надо включать в wav-файл с выделяемым блоком; они не несут полезной для нас информации, и являются помехой для наших раскодировочных процедур.

Просматривая картину, надо также обращать внимание на дефекты сигнала; пример:



С дефектными блоками не стоит связываться. Из трёх копий блока надо выделять наилучшую.

б) Начало в маткадном документе для раскодировки 256-байтного блока точно такое же, как и рассмотренное выше для обычной программы произвольной длины (поэтому здесь его не воспроизвожу; см. описание процедур и построения графиков на стр. 3–14; о дифференцировании «П-образных» импульсов рассказано на стр. 26).

Различие (не считая разницы в именах загружаемых wav-файлов и в настройке параметров D0min, D1min) появляется только после вызова функции BitStream, т. е. после строк

$$Bits1 := BitStream(Diff2)$$
$$Nbits1 := length(Bits1)$$

Поскольку в 256-байтном блоке отсутствуют контрольные биты чётности («девятые биты»), то теперь проверка количества байт в намечающейся раскодировке даётся делением $Nbits1 - 1$ не на 9, а на 8; соответственно, деление на 8 присутствует в определении переменной Num2:

$$Num2 := trunc\left(\frac{Nbits1 - 1}{8}\right)$$

Вызывать функцию Bits2(Bits1), проверявшую чётность и удалявшую девятые биты, теперь не надо. Массив Bits1 сразу поступает в роли аргумента в функцию Bytes1 (такую же самую, как и при раскодировке обычной программы, см. стр. 19), которая разбивает его на байты:

$$BytesBlock := Bytes1(Bits1)$$

Массив байт-кодов BytesBlock – это и есть раскодировка 256-байтного блока, но она пока ещё непроверенная.

в) Микропрограмма чтения с МЛ 256-байтных блоков в машине Д3-28 никак не проверяет правильность чтения. Организация контроля возложена на программиста и, к сожалению, не подчиняется какому-либо стандарту. Поэтому определять способ вычисления контрольной суммы и место её расположения в 256-байтном блоке приходится экспериментально, либо – путём изучения дизассемблированного кода загрузчика.

В рассматриваемом примере изучение кода 1-го загрузчика показывает, что правильность раскодировки 2-го загрузчика можно проверить обычным подсчётом контрольной суммы (обычным суммированием десятичных значений тетрад) по байтам с номерами 0, 1, … , 251. Процедуру такого подсчёта контрольной суммы (КР) мы рассмотрели в разделе **7** на стр. 20. В ней номера начального и конечного байтов для подсчёта КР задаются в переменных N1 и N2. Однако для сравнения с ожидаемым значением в виде двух байтов 0015 0404 мы должны перевести десятичное значение КР в двухбайтовую тетрадно-десятичную форму. Вот процедура перевода КР в такие четыре тетрады (обозначенные как P3, P2, P1, P0):

$$P3 := trunc\left(\frac{KP}{4096}\right)$$
$$P2 := trunc\left(\frac{KP - 4096 \cdot P3}{256}\right)$$
$$P1 := trunc\left(\frac{KP - 4096 \cdot P3 - P2 \cdot 256}{16}\right)$$
$$P0 := KP - 4096 \cdot P3 - P2 \cdot 256 - P1 \cdot 16$$

Вот вывод их в документ в двухбайтовой форме, а также вывод четырёх последних байт и вывод значения $(Nbits1 - 1) / 8$ (чтобы убедиться, что оно целочисленное):

$BytesBlock_{252} = 0$

$BytesBlock_{253} = 0$

$P3 \cdot 100 + P2 = 15$ $BytesBlock_{254} = 0$

$P1 \cdot 100 + P0 = 404$ $BytesBlock_{255} = 0$ $\frac{Nbits1 - 1}{8} = 256$

Как видим, количество раскодированных байт в загрузчике-2 получилось верное (256) и контрольная сумма совпала с ожидаемой (0015 0404). В таком случае выполняем итоговый шаг – запись этой раскодировки в свой отдельный txt-файлик:

file2 := "c:\L_Work\test-progs\for_d3-28\k16a_vt-9r_zagr-2_t12.txt"
WRITEPRN(file2) := BytesBlock

Затем аналогичным образом выполняем раскодировку первого 256-байтного блока из серии основных блоков программы (из серии 54 блоков с программой ВТ-9Р в данном примере). Вывод проверочной информации для этого блока при подсчёте КР по байтам с теми же номерами, что и в предыдущей раскодировке, выглядит так:

$BytesBlock_{252} = 8$

$BytesBlock_{253} = 408$

$P3 \cdot 100 + P2 = 14$ $BytesBlock_{254} = 14$

$P1 \cdot 100 + P0 = 407$ $BytesBlock_{255} = 511$ $\frac{Nbits1 - 1}{8} = 256$

Видно, что подсчитанная так КР немножко не совпадает с последними двумя байтами в блоке. Догадываемся, что здесь следует поэкспериментировать с номерами байтов для подсчёта КР. И действительно, заменив значение N2 : = 251 значением N2 : = 253, уже получаем совпадение:

$P3 \cdot 100 + P2 = 14$ $BytesBlock_{254} = 14$

$P1 \cdot 100 + P0 = 511$ $BytesBlock_{255} = 511$ $\frac{Nbits1 - 1}{8} = 256$

Тогда выполняем итоговый шаг – записываем эту раскодировку в свой отдельный txt-файлик. Таким образом, мы экспериментально нашли способ вычисления контрольной суммы и место её расположения в основных 256-байтных блоках в данном примере.

И так далее – указанным методом раскодируем и записываем в отдельные txt-файлы остальные 53 блока программы ВТ-9Р. Если в процессе раскодировки встречается ошибка – получается неверная КР или нецелое либо отличное от 256 количество байт, – то приходится отыскивать сбойное место просмотром графиков нормализованных семплов совместно с wav-данными по всему файлу (так как в 256-байтном блоке нет контроля чётности, то нет возможности автоматически определять район ошибки). О способах исправления ошибок уже рассказывалось в первой части; к 256-байтным блокам примен*и*м аналогичный подход.

Завершающее действие – сборка отдельных txt-файлов с помощью текстового редактора в единый txt-файл. В нём для запуска в имитаторе Д3-28 начало и конец каждого 256-байтного блока отмечается пятнадцатью символами нижнего подчёркивания (для примера см. готовые образцы раскодировок).

31.05.2019. Sinus.