FAQ (ЧаВО) по PROTEUS для начинающих и не только.

ЧАСТЬ IV. PROTEUS для фанатов – продолжение.

Содержание: *Стр.*

7. Активные модели.

7.1. «Всё смешалось в доме Облонских…» или небольшое лирическое отступление о том, что считать активным в Протеусе.

Когда то в очень отдаленном прошлом, в эпоху диодов серии Д2 и транзисторов П4Б было принято называть активными компоненты (или как тогда гласила терминология радиоэлементы), которые производят над электрическим сигналом определенные преобразования: усиление, выпрямление и т.п. и т.д.. Однако, прогресс не стоит на месте, и в нынешнюю эпоху повальной компьютеризации несознательного населения устоявшиеся термины приобретают совсем иное значение. И некогда считавшийся пассивным элемент, тот же резистор может с успехом при компьютерном моделировании оказаться намного «активнее» самой навороченной микросхемы. Давайте сразу расставим все на свои места, чтобы не путаться с терминологией в дальнейшем материале.

В случае компьютерного схемотехнического моделирования, как и в любой другой компьютерной программе под активностью подразумевается поведение объекта. Вспомним те же элементы **ActiveX**, применяемые в **Windows** – ползунки, движки, всплывающие меню, прогресс-бары и прочую мерцающую мишуру. Именно их поведение на экране и возможность пользователя активно вмешаться в процесс: сдвинуть, увеличить или уменьшить определяет их как активные. С той же точки зрения надо рассматривать и активные модели в схемотехническом моделировании и в частности в Протеусе. Это те модели, которые либо меняют свое отображение на экране – индикаторы, либо позволяют вмешаться в процесс симуляции – модели, имеющие актюаторы. Вот их мы и будем далее рассматривать в этом разделе.

В **ISIS**, по сравнению с другими пакетами моделирования, и сейчас достаточно много уже существующих активных моделей, способных удовлетворить даже привередливого пользователя. Тут и LED и LCD индикаторы и всевозможные кнопки, переключатели, реле и даже различные датчики физических величин и моторы. Но нашему пытливому русскому уму этого мало. В этом разделе мы будем вести себя как малые дети – попытаемся заглянуть, а что внутри у заморской куклы Барби. Все переломаем и соберем по своему, тут главное не напороть горячки и не «пришпандорить к гульфику рукав», как в известной интермедии Аркадия Райкина. Поэтому, призываю быть особенно внимательными при повторении или самостоятельном «изготовлении» активных моделей. Порой пропущенная буква или неправильно поставленный маркер могут привести к совершенно непредсказуемым результатам и полной неработоспособности вашего творения. Поэтому этот раздел действительно для фанатов, способных часами «отлаживать» поведение модели на экране компьютера. На наиболее значимых моментах я постараюсь останавливаться неоднократно, как делал это ранее с той же процедурой **Make Device**, которая по-прежнему останется для нас основной и самой применяемой. В основном, рассматриваемый далее материал полный «эксклюзив», давшийся мне, как говорится – «потом и кровью». Нигде в HELP Протеуса и сторонних публикациях это не описано и найдено методом «ненаучного», но все же интуитивно предсказуемого «тыка». На этом мое лирическое отступление заканчивается и начинается, надеюсь самый интересный и полезный раздел FAQ по Протеусу.

К содержанию

7.2. Снова в 2D графику. Графические символы – основа активных моделей. Маркер ORIGIN – Архимедова «точка опоры» для активной графики.

До сей поры, мы использовали ту часть левого меню, которая относится к **2D** или, по-русски говоря плоской двухмерной графике, поскольку трехмерная в **ISIS** не предусмотрена, только для прорисовки графического изображения модели. Надеюсь, больших затруднений рисование линий, прямоугольников и кругов у вас не вызвало. Несколько сложнее нарисовать с помощью **2D Graphic Closed Path** замкнутую многоугольную фигуру, ну и практически совсем невозможно изобразить замкнутую криволинейную – нет, к сожалению, такой опции в графическом арсенале **ISIS**. Ну, что-же, будем довольствоваться тем, что есть. В конце концов это не **AutoCAD** и не **PhotoShop**, у Протеуса совсем другие задачи и «Джоконду» или «Девочку с персиками» нам тут рисовать не потребуется. Хотя, как раз в графических символах, порой приходится обходиться упрощенными изображениями там, где явно не хватает как раз замкнутой залитой цветом криволинейной фигуры.

Итак, что же такое графический символ в **ISIS** и чем он отличается от обычного графического изображения модели компонента. Давайте еще раз вспомним, как мы рисовали компонент. На поле проекта рисовался **2D Graphic** базовый прямоугольник (квадрат, треугольник, круг или что-то еще) – тело компонента, как правило в стиле **Component** (коричневое обрамление и телесная заливка)**.** К нему пристыковывались выводы **Pins** из левой колонки меню **Device Pins Mode**, а также ставился один маркер привязки, называемый **ORIGIN**. Внутри или снаружи тела можно было дорисовать линиями или арками еще «что-нибудь ненужное», например знаки полярности или линии раздела, а

также нанести в стиле 2D надписи (Рис. 1). На этом процесс рисования заканчивался, и мы приступали к операции **Make Device** для создания графического изображения модели компонента.

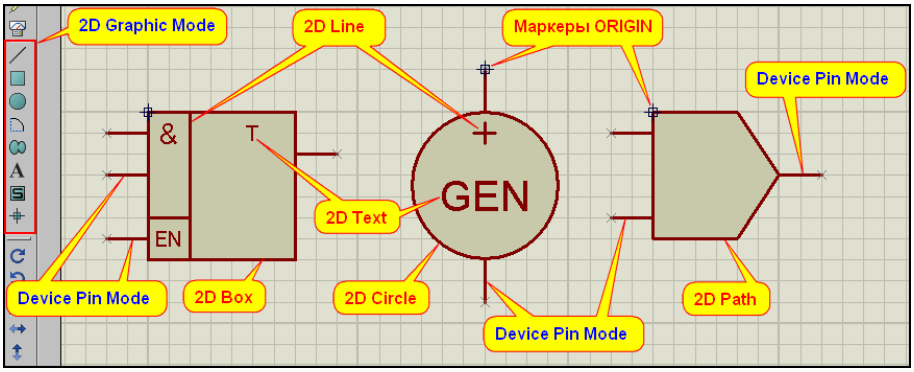


Рис. 1

Изображение для компонента у нас всегда было одно, поскольку оно единожды, раз и навсегда вставлялось в проект и в этом виде присутствовало там постоянно.

С графическими символами немного иначе. Фактически, это часть изображения компонента, которая может изменять свой вид, цвет, положение или числовое значение в процессе выполнения симуляции. Если кто-то увлекался программированием графики, это как-бы спрайт – есть такой термин у программистов графики. В процессе создания активного компонента символы интегрируются в модель (в ту часть, которая располагается в папке **LIBRARY**) и затем, в процессе симуляции воспроизводятся на экране в зависимости от наличия определенных сигналов на выводах или внутреннего состояния модели. Создание символа практически ничем не отличается от создания графического изображения компонента, но все же есть один существенный нюанс, связанный с маркером **ORIGIN**. В компоненте мы его обычно ставим либо в верхнем левом углу тела компонента, либо привязываем к концу одного из выводов (Рис. 1). В одном из начальных разделов я уже упоминал, что он служит для выравнивания компонента по координатной сетке. Но второе его назначение тогда мы не рассматривали, а состоит оно в том, что он служит еще и отправной начальной точкой, относительно которой выравниваются графические символы в активных компонентах. В качестве примера давайте разберем на составные части какой-нибудь семисегментный индикатор, например, модель **7SEG-COM-ANODE**. Весь это проект с комментариями представлен в проекте **7_SEG.DSN** вложения. Помещаем модель в поле проекта и, выделив, выбираем опцию **Decompose** либо через меню правой кнопки мыши, либо через верхнее меню – кнопка с изображением молотка. Теперь в левом вертикальном тулбаре выбираем режим отображения и редактирования символов – латинская **S** в квадратике. После этого в селекторе объектов мы увидим список всех символов, входящих в модель **7SEG-COM-ANODE** (Рис. 2).

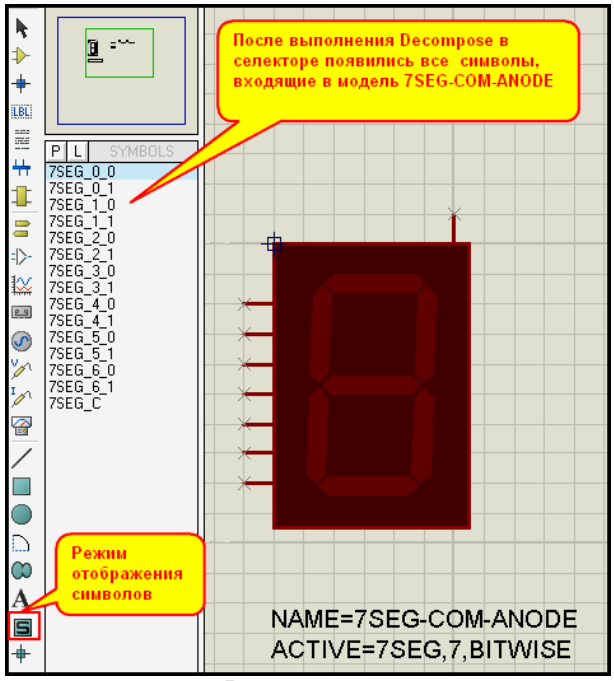


Рис. 2

Само разобранное графическое изображение в поле проекта интересует нас в данный момент только с точки зрения расположения маркера **ORIGIN** – верхний левый угол тела (прямоугольника) модели. Именно он является «базовой точкой» для всех символов, входящих в модель. Если выбрать в селекторе какой либо символ, то в окне предпросмотра вы увидите его вместе с входящим в символ маркером ORIGIN (Рис. 3). В качестве примера на этом рисунке я выделил символ **7SEG_0_1** – светящийся сегмент **«a»** семисегментного индикатора.

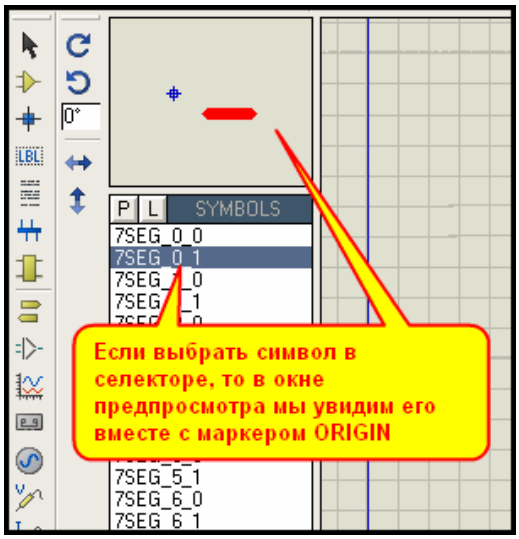


Рис. 3

Однако если вы, выбрав какой либо символ, кликните левой кнопкой мышки в поле проекта и установите его (установится он, как и компонент, вторым кликом по левой кнопке), то маркер в поле проекта будет не виден. Все дело в том, что любой графический символ сам является сложным объектом, состоящим из изображения и маркера **ORIGIN** и так же, как любая модель подлежит предварительной компоновке. Только делается это не через **Make Device**, а через **Make Simbol**. Эта опция отсутствует в верхнем и боковых тулбарах и доступна только через меню правой кнопки мыши (Рис. 4).

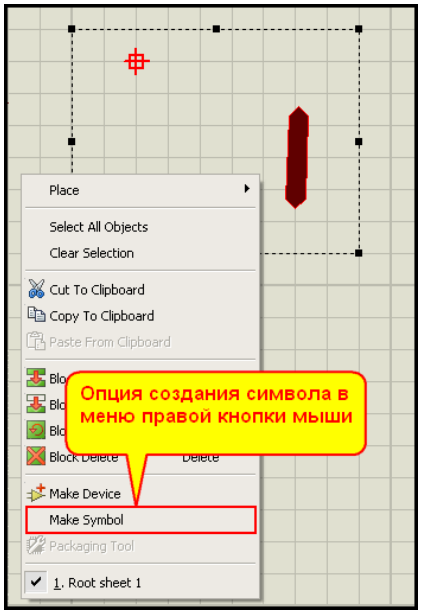


Рис. 4

Соответственно, раз мы создаем символ через **Make Simbol**, то мы можем применить к нему и «разборку на запчасти», т.е. опцию **Decompose**. Вот тогда-то мы и увидим отдельно графику и отдельно маркеры (Рис. 5).

Рис. 5

На что здесь хотелось бы обратить ваше внимание. Если выделить разобранный символ сегмента вместе с маркером и переместить на разобранное графическое изображение компонента так, чтобы маркеры сегмента и всего компонента совпали, то мы увидим, что и сегмент окажется точно на том месте, где он расположен на изображении компонента. Вот для этого и служит маркер **ORIGIN** в символе. Он точно позиционирует графическое изображение символа на изображении всего компонента в момент симуляции. Если не соблюдать этого простого, но очень строгого правила привязки, то при запуске симуляции сегменты вашего активного компонента будут съезжать относительно основного изображения или хаотично прыгать по экрану. Это первый признак того, что вы допустили ошибку при компоновке графических символов.

Теперь разберем состав и нумерацию графических символов на примере все того же семисегментника **7SEG-COM-ANODE**. Если внимательно глянуть в селектор на рисунке 2, то можно заметить некоторую закономерность. Все символы (сегменты) имеют в начале имени аббревиатуру **7SEG**. Совсем не обязательно привязывать данную часть именно к назначению нашего компонента. С тем же успехом, наши символы-сегменты могли бы именоваться и **ABC** и **123A** и вообще так, как подсказывает ваша фантазия. Тут главное, чтобы у всех символов, принадлежащих одному компоненту, эта часть наименования совпадала. Кроме того, настоятельно рекомендую использовать только латинские символы и цифры и не пользоваться спецсимволами и знаками препинания. В частности, знак подчеркивания в наименовании символа служит для разделения частей имени и применение его в других местах приведет к неработоспособности модели. За начальным именем через знак подчеркивания следует номер графического символа. Так как мы разбили семисегментный индикатор, то и таких номеров у нас будет семь, начиная с нулевого и заканчивая шестым. Для каждого символа сегмента в данной модели определено два состояния – погасший и светящийся. Эти состояния указаны через еще один знак подчеркивания. Погашенному состоянию соответствует цифра 0, а светящемуся – 1. Таким образом, полное наименование символа для погашенного сегмента **«a»** будет **7SEG_0_0**, а для засвеченного **7SEG_0_1**, а, например, для сегмента **«d»** (нижний горизонтальный) соответственно **7SEG_3_0** и **7SEG_3_1**. Отдельно стоит остановиться на символе **7SEG_C**. Если выделить его в селекторе, то в окне предпросмотра мы увидим, что он состоит из основного графического изображения – «тела» компонента и маркера **ORIGIN**. Он является базовым, как бы подложкой для наших символов-сегментов во время симуляции и на его фоне мы и будем наблюдать изменение состояния наших сегментов на экране. Для модели **7SEG-COM-ANODE** он представляет из себя базовый прямоугольник с маркером **ORIGIN** (Рис. 6).

Рис. 6

На рисунке 6 представлены все символы, входящие в модель **7SEG-COM-ANODE** в «разобранном» (**Decompose**) виде. Я специально их «разобрал» в поле проекта, чтобы видно было положение маркера **ORIGIN** и смещение самого изображения сегмента относительно этого маркера. Еще раз подчеркну, что поскольку в данном случае мы имеем дело с так называемым **Bitwise** индикатором, то для каждого сегмента определена группа из двух символов неактивный (в конце символ 0) и активный (в конце символ 1). Термин **Bitwise** дословно означает «зависимый от бита». В данном случае текущее состояние символа **7SEG**_x (где x номер символа) зависит от логического состояния на определенной ножке (**pin**) модели – выводы A, B … G. Для модели с общим анодом это означает, что если там находится логический ноль, то сегмент должен светиться – отображается символ **7SEG**_x_**1**, если на выводе логическая единица – сегмент не светится – отображается **7SEG**_x_**0**. Здесь мы имеем дело с цифровым типом индикации (хотя позже я покажу, что на самом деле и не совсем цифровым), но символы совсем не обязательно могут иметь только два вида. Давайте для примера «разберем на запчасти» модель обычного светодиода **LED_GREEN**. Этот пример с комментариями представлен в проекте **LED.DSN** вложения. Попутно хочу дать полезный совет. Проводя различные исследования с активными элементами в ISIS, старайтесь не увлекаться «разборкой» нескольких моделей в одном проекте. Иначе селектор символов станет похож на отхожее место роты солдат в полевом лагере. Дерьма много, а где чье – непонятно. А в ряде случаев может оказаться, что символы одной модели имеют то же имя и заменят символы другой.
На рисунке 7 представлены символы входящие в модель **LED_GREEN**. Как видим, здесь в нумерации используется только одна цифра – номер символа от 0 до 7. Соответственно и символы имеют наименование **LED_GREEN_0** – полностью погашенный светодиод, **LED_GREEN**_1 – слегка подсвеченный и, наконец, **LED_GREEN_7** – полностью светящийся.

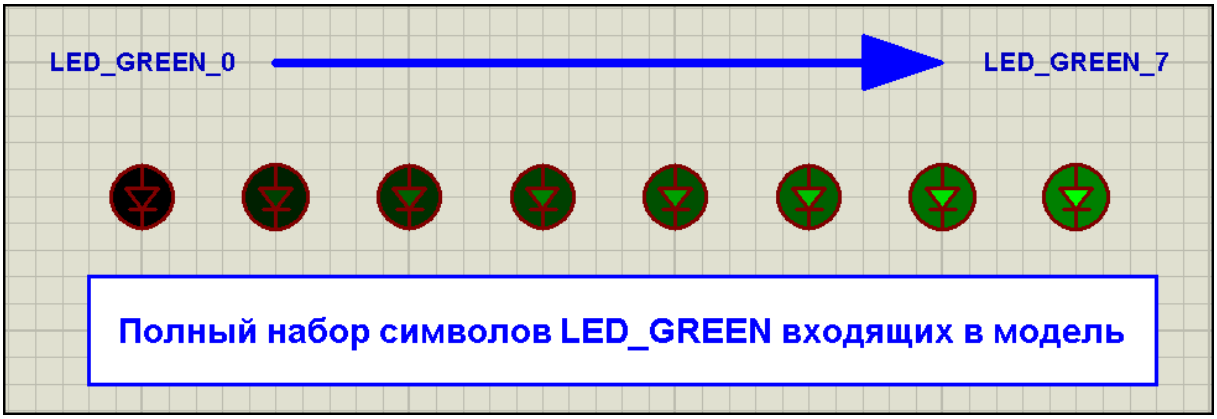


Рис. 7

Всего таких состояний в обычных активных индикаторах может быть 32 (нумерация от 0 до 31). На это тоже прошу обратить особое внимание. Это относится и к **Bitwise** индикаторам, но только в том случае, если мы не используем специализированные DLL-библиотеки для их создания. Там суммарное количество элементов определяется самой применяемой DLL и об этом речь пойдет позже.
В модели **LED_GREEN** сами символы имеют более сложную структуру из нескольких графических элементов и маркера **ORIGIN**. На рисунке 8 представлен «декомпозированный» символ **LED_GREEN_7**. Справа на рисунке все составляющие символа с указанием того в каком режиме рисуется данный элемент.

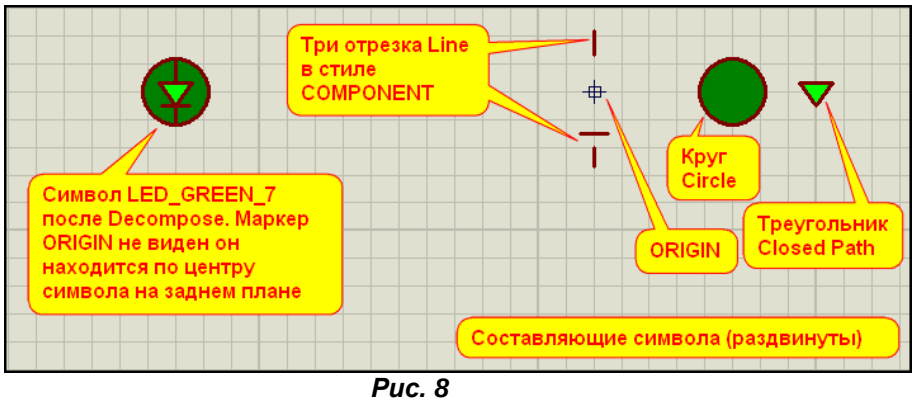


Рис. 8

Как видим, здесь мы имеем дело с более сложным по наличию графических элементов символом. И снова хочу заострить ваше внимание на этой графике, поскольку при создании таких элементов легко можно допустить характерную ошибку. Связана она с расположением графических элементов «в глубину» экрана. ISIS имеет только две опции, расположенные в верхнем меню **Edit**:
Send to back (CTRL+B) – отправить элемент на задний план.
Bring to front (CTRL+F) – выдвинуть элемент вперед.
Этими опциями надо умело пользоваться, иначе вы рискуете получить при создании символа (да и компонента тоже) «пропадающие» с экрана элементы графики. В случае символа светодиода должны быть выдвинуты на передний план отрезки и светящийся треугольник. На самом заднем плане окажется маркер **ORIGIN**, расположенный в данном случае по центру символа.
На этом, пожалуй, можно закончить наше знакомство с особенностями создания графических символов для активных моделей. Настала пора позаботиться о том, где сохранить плоды наших «графических фантазий». И здесь разработчики программы позаботились о нас заранее, но об этом в следующем материале.
Еще одно небольшое замечание по вложению. Подсветка имени выводов моделей при сохранении проекта теряется, поэтому если Вы желаете видеть ее на экране, то включите ее самостоятельно либо в свойствах соответствующих **pin** (выводов), либо через **PAT** (что быстрее). В окне **Sting PAT** набираем **NAME**, ставим переключатели **Action** => **Show** и **Apply To** => **All Objects**, жмем **OK** и получаем нужное. Привыкайте работать «с удобствами».
Внимание. *Начиная с этого раздела в **OnLine** версии имена архивов вложений как и в **OffLine**: номер раздела – символ подчеркивания – номер параграфа. Соответственно здесь: 7_2.rar.*

К содержанию

7.3. «Символические» библиотеки. Использование менеджера библиотек для библиотеки символов.

Мы уже давно привыкли пользоваться библиотеками компонентов для подбора нужных нам в конкретном проекте. Как я уже подчеркивал, для этого всего-навсего необходимо дважды щелкнуть левой кнопкой мышки в свободном поле селектора или нажать в нем кнопочку **P** в левом верхнем углу. При этом многие уже попадали в библиотеку символов по неведению, но закрывали ее «дабы от греха подальше». Теперь мы сознательно заглянем туда. Проще всего попасть в нее, находясь в режиме **2D Graphic Symbols Mode** (нажата кнопочка **S** в левом тулбаре). Процесс аналогичен визиту в библиотеку компонентов – двойной клик левой в селекторе, или одинарный по буковке **P**. Обратите внимание, что при этом справа от кнопок **P** и **L** вверху окна селектора стоит серая подсказка **SYMBOLS**. После этого откроется окно для выбора библиотеки и символов в ней (Рис. 9).

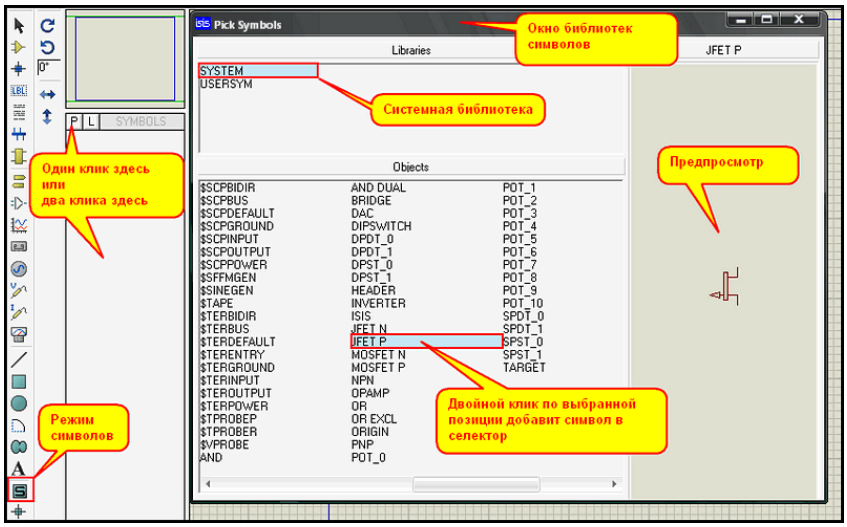


Рис. 9

Картинка немногим отличается от окна библиотеки компонентов, да и операции схожие. В разделе Libraries выбираем нужную библиотеку. Пока их всего две: **SYSTEM** – системная библиотека символов и **USERSYM** – пользовательская, которая, если мы еще не создавали собственных символов, абсолютно пустая. В разделе **Objects** выбирается требуемый символ, который отображается справа в окне предпросмотра. Двойным щелчком мышки мы можем добавить его в селектор символов **ISIS**. Сразу обращаю ваше внимание, что системная библиотека символов не защищена от записи по умолчанию, поэтому если вы сохраняете свой символ, то в первую очередь **ISIS** предложит сохранить в ней. Чтобы не создавать там каши из системных символов и своих собственных будьте внимательнее, или защитите ее от записи в менеджере библиотек. Давайте теперь заглянем в сам менеджер библиотек и попутно создадим собственную библиотеку

для хранения графических символов. Для этого в режиме символов достаточно кликнуть мышкой по кнопочке **L** вверху окна селектора или из этого же режима зайти в **Library Manager** через верхнее меню **Library**. При этом откроется уже знакомое нам окно (Рис. 10). Однако если в нем мы будем выбирать через раскрывающиеся списки **Source** (источник) или **Dest'n** (приемник), то доступны будут только все те же **SYSTEM** и **USERSYM**. Вот здесь и можно запретить запись символов в библиотеку **SYSTEM**. Не забудьте предварительно кликнуть мышкой по той части окна, где она находится. Потом нажимаем кнопку **File Attribute** и на вопрос Протеуса, установить ли режим **Read Only** для выбранной библиотеки нажимаем **OK**. После такой процедуры данная библиотека даже не будет предлагаться для сохранения символов.

Ну а теперь создадим свою библиотеку для символов. В принципе, можно пользоваться и **USERSYM**, но я сторонник крайностей – как поется в популярной ныне на Радио-Шансон песенке: «лучше маленький, но свой». Я уже пояснял по этому поводу при создании компонентов, но здесь вопрос стоит еще острее. Дело в том, что красивые графические символы требуют на свое создание гораздо больших затрат времени, чем сирые и убогие компоненты. И уж куда обиднее будет потерять «нажитое непосильным трудом» при переустановке Протеуса, или копировании чужой **USERSYM** поверх своей. Так что, к делу…

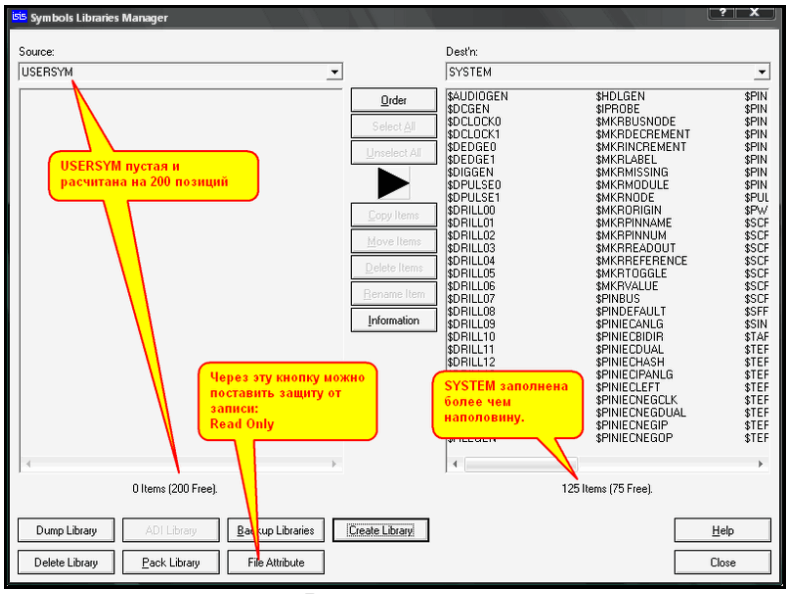


Рис. 10

В окне менеджера библиотек нажимаем кнопку **Create Library** и попадаем в окно выбора папки для сохранения (Рис. 11). Обратите внимание, что по умолчанию нам предлагается сохранить библиотеку в папке **LIBRARY** установленного Протеуса. Папку менять не будем, а вот название зададим свое, например, **MY_SYMBOLS**.

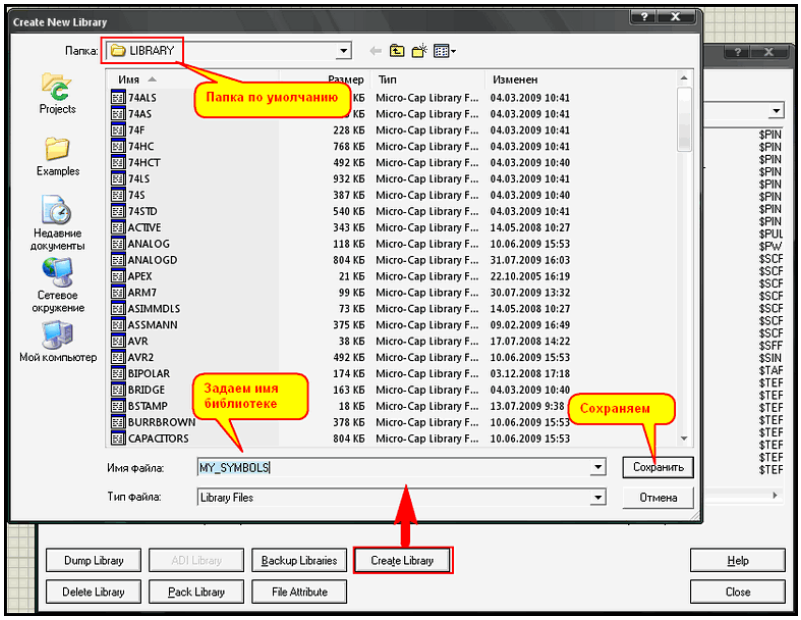


Рис. 11

После нажатия на кнопку: «**Сохранить**» нам будет предложено определить количество хранимых в библиотеке символов – **Maximum Entries**. Здесь ужиматься не стоит, поскольку количество собственной графики будет возрастать семимильными шагами. Поэтому я задал стандартное для библиотек символов количество – 200 и подтвердил создание кнопкой **OK**. После этого менеджер библиотек можно покинуть через кнопку **Close**, так как библиотеку мы уже полностью создали.
Если теперь, находясь в режиме отображения символов вновь дважды кликнуть в селекторе, то в открывшемся окне будут доступны для выбора уже три библиотеки, в том числе и наша «свежеиспеченная» **MY_SYMBOLS** (Рис.12). Конечно, пока она абсолютно пустая, но в скором времени мы заполним ее так, что еще и места будет мало. И начнем мы это делать прямо сейчас. Для начала поучимся создавать простые активные индикаторы на примере все того же семисегментника, только раскрасим его по своему и прилепим туда недостающую десятичную точку.

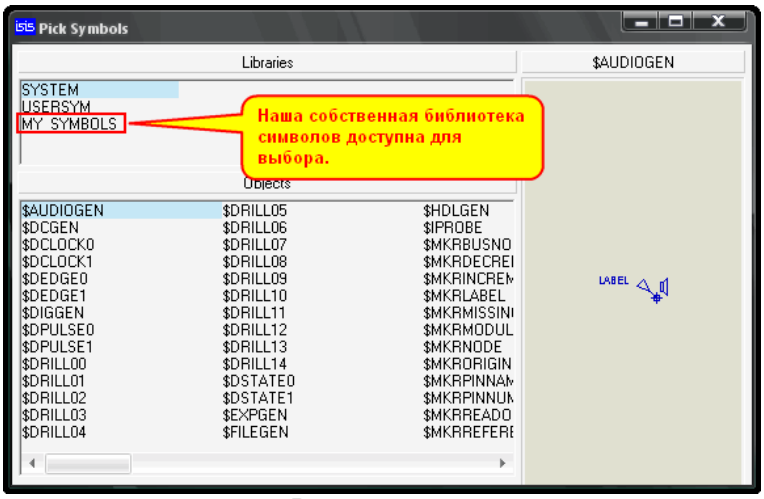


Рис. 12

К содержанию

7.4. Пример создания активного семисегментного индикатора с десятичной точкой. Часть первая – графика.

В качестве первого пробного шара в создании собственных активных компонентов я решил использовать все тот же семисегментный индикатор с общим анодом. На его основе мы поучимся создавать простые **Bitwise** индикаторы. Не будем особо усложнять задачу, а просто сделаем индикатор желтого цвета – таковой отсутствует в стандартной библиотеке и попутно приклеим к нему десятичную точку. Для этого нам потребуется уже разобранный ранее **7SEG-COM-ANODE**. От него мы используем существующую графику, только перекрасим ее. Кроме того, если вы обратили внимание, этот индикатор позиционируется как **Schematic Model**, следовательно, для него имеется файл **MDF**. Называется он **7SEGCOMA** и расположен в библиотеке **DISPLAY.LML**. Как его добыть оттуда, вы уже знаете, повторяться не буду. Я просто приложил уже извлеченный в папке **Prototip** вложения. В довесок там же **7SEGCOMK.MDF** для индикатора с общим катодом и **LED.MDF** для активного светодиода из библиотеки **ACTIVE.LML** для тех, кто хочет поупражняться самостоятельно в создании индикаторов. Итак, приступим.
Начало процесса не вызывает особых затруднений. Мы просто помещаем в новый проект модель **7SEG-COM-ANODE** и применяем к ней **Decompose**. Теперь мы имеем разобранный на запчасти индикатор и кучку символов в селекторе символов (режим **Simbol Mode**). Их мы тоже извлекаем в проект все по порядку и «разбираем на запчасти» через **Decompose**, чтобы графика отдельно - **ORIGIN** отдельно. После этого селектор символов можно очистить, чтобы не путаться со старыми символами. Делается это, как и с компонентами – щелкаем правой кнопкой мышки в селекторе и выбираем опцию **Tidy**. Селектор чист, и можно приступить к созданию собственных символов. На этом первая стадия нашего эксперимента закончена, и я сохраню ее в этом виде в папке **Begin** вложения.
Приступаем ко второй стадии. Для начала перекрасим светящиеся и погашенные сегменты. Дважды щелкаем по первому темнокоричневому сегменту «**a**» и в открывшемся окне Edit path's graphic style выбираем любой из **Colour**, а в открывающемся дополнительном селекторе кнопку **other…**. Для погашенных элементов я выбрал один из темных оттенков желтого цвета и добавил его в набор (Рис. 13).

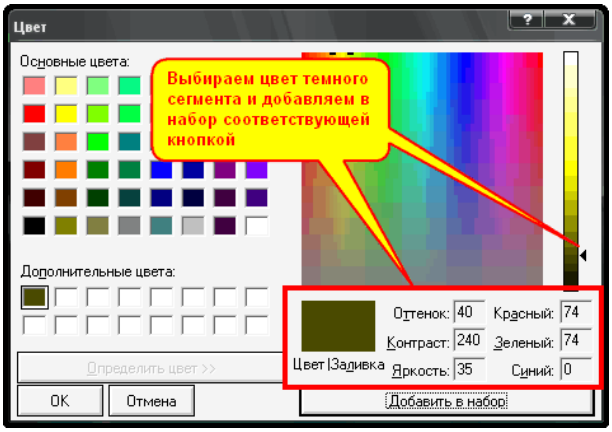


Рис. 13

Для того, чтобы перекрасить весь сегмент нам необходимо выбрать одинаковый цвет как для линии обводки (бордюра), так и для заливки фигуры (Рис. 14). После этого нажимаем кнопку **This Graphic Only**, чтобы сохранить изменения. Проделываем указанную процедуру со всеми погашенными сегментами.

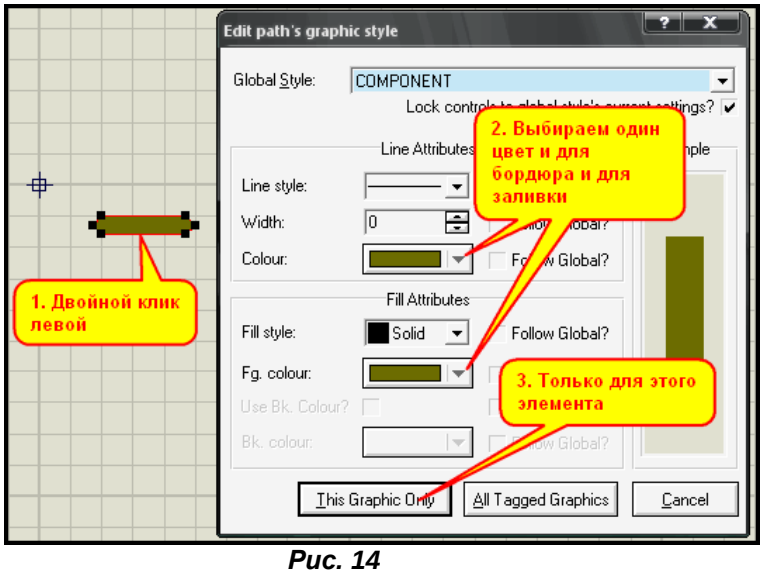


Рис. 14

Для засвеченных сегментов аналогично выбираем стандартный ярко-желтый цвет из палитры основных цветов: второй ряд – второй слева цвет. Кроме того, нам необходимо перекрасить в цвета погашенных сегменты на основном изображении компонента (Рис. 15).

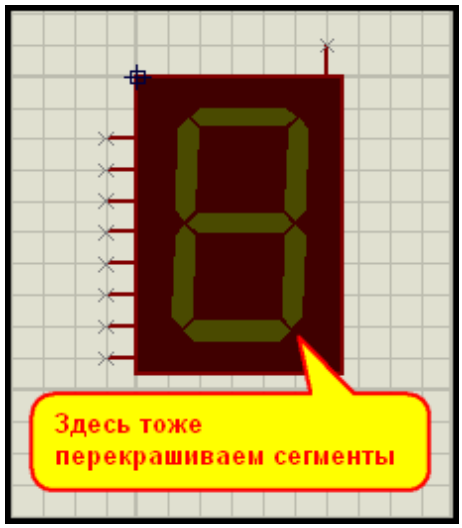


Рис. 15

Тело компонента, а также бывшая и будущая подложка с индексом _C на конце в покраске не нуждаются, хотя наиболее привередливые могут изменить цвет и у них, например, на стандартный для реальных семисегментников серый. Тут главное изменить цвет и у компонента и у подложки на один и тот же. На этом «малярные» работы заканчиваются, и мы приступаем к созданию своих символов.

Создание символов наиболее ответственный момент во всей процедуре, и тут нужно повышенное внимание, чтобы не напороть косяков. Выделяем первый по счету сегмент обязательно с принадлежащим ему маркером **ORIGIN** и через правую кнопку мышки выбираем опцию **Make Symbol** (Рис. 16).

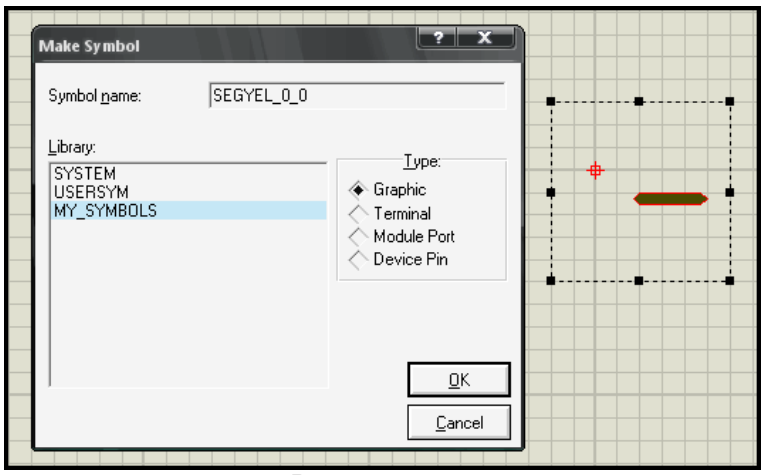


Рис. 16

Обратите внимание, что библиотека **SYSTEM** у меня не защищена и присутствует в окне выбора **Library**. Я достаточно уверен в своих действиях, как шутят автомобилисты – «мастер за рулем», поэтому сразу же выбрал библиотеку **MY_SYMBOLS**. В переключателе **Type** оставляем предлагаемый по умолчанию **Graphic** и задаем имя первого нашего символа. Я назвал его **SEGYEL_0_0** по аналогии с разбиравшимся выше.

Здесь сразу хочу предложить одну «фишку» собственного пошиба. Чтобы каждый раз не набирать имя вручную – не торопитесь давить кнопку **OK**. После набора полного имени выделите его и скопируйте в буфер обмена через CTRL+C или правую кнопку мышки – **«Копировать»**. При создании следующего символа достаточно в этом поле вставить из буфера готовое имя и поправить всего одну или две цифры. Этим вы по крайней мере гарантируете себя от ошибок в наборе имени, ну и чуть-чуть ускорите сам процесс. Есть еще один «неприятный нюанс» Протеуса. После создания каждого символа необходимо в левом тулбаре переключаться из режима символов в режим **Selection Mode** (самая верхняя кнопка с жирной косой стрелкой). Иначе курсор будет находиться в режиме рисования (карандаш) и вы не сможете выделить следующий символ. Следующим по счету будет светящийся желтый символ сегмента «**a**», а имя у него будет отличаться только последней цифрой **SEGYEL_0_1**. Третьим будет темный символ сегмента «**b**» с именем **SEGYEL_1_0** и т.д. до заключительной подложки с именем **SEGYEL_C**. В финале селектор символов у вас будет выглядеть так, как представлено на рисунке 17.

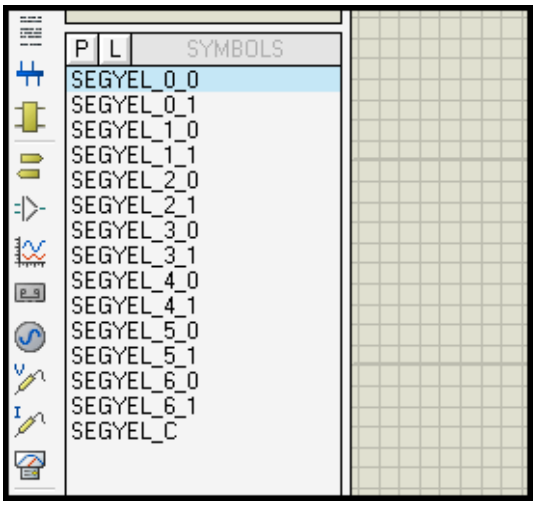


Рис. 17

Ну а если теперь заглянуть в нашу библиотеку символов, то можно убедиться, что все созданные нами символы представлены и там (Рис. 18).

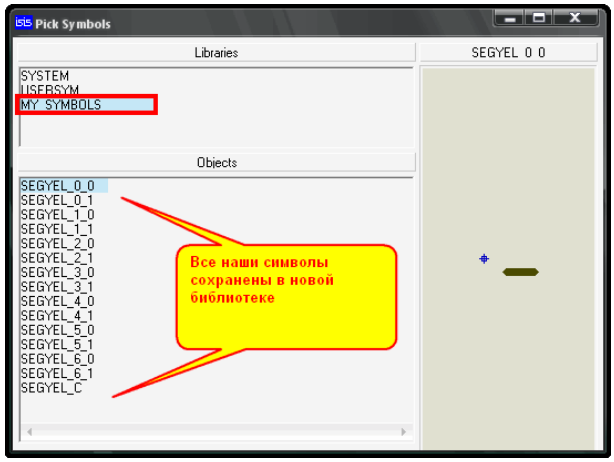


Рис. 18

Но мы собирались добавить к нашей модели еще и десятичную точку. Пора заняться и этим. Проще всего пририсовать для начала точку на полной модели с выводами. При этом у нас будет полная ориентация относительно всех сегментов и края тела модели. Здесь временно потребуется переключиться через меню **View** в режим сетки **Snap 10th**, иначе нам не удастся соблюсти нужные зазоры и размер точки. Кроме того, размер бордюра у нарисованной окружности необходимо установить в 0, как и у сегментов, ну и конечно выбрать для бордюра и заливки цвет погашенного сегмента. Сразу же добавим и необходимый вывод со скрытым именем **H** (Рис. 19). Теперь копируем полностью графическую модель на свободное место в проекте через **Block Copy** в двух экземплярах и удаляем на ней все лишнее, кроме самой точки и маркера **ORIGIN**. Таким «хитрым способом» мы точно обеспечим соблюдение смещения точки относительно маркера **ORIGIN**.

Рис. 19

Нам осталось изменить цвет для светящейся точки и создать два дополнительных символа с именами **SEGYEL_7_0** и **SEGYEL_7_1** для погашенной и светящейся точки, как мы делали ранее.
На этом процедура создания графики для нашего семисегментного индикатора завершена. В следующем параграфе займемся созданием полной модели с пробной симуляцией.

К содержанию

7.5. Пример создания активного семисегментного индикатора с десятичной точкой. Примитивы для создания индикаторов. Часть вторая – модель.

Для начала предстоит создать графическую модель компонента, которая и будет представлять его в библиотеке, но попутно мы включим туда и свойства активного компонента. Итак, графическую (не симулируемую) модель создаем из того набора, что представлен на предыдущем рисунке вместе с добавленным выводом и десятичной точкой. Как и обычно, выделяем все это рамочкой и нажимаем нашу любимую кнопку **Make Device**. Именно девайс, символов мы уже насоздавались. И вот тут с

самой первой вкладки начинаются новшества. Я уже упоминал, что мы создаем бит-зависимый активный компонент. Зададим ему имя, пусть он **7SEG-COM-ANODE-YEL** (больше просто не упишется), префикс можно и не задавать, тут это не критично и делается как всегда.

Приключения начинаются в нижней части окна – **Active Component Properties** – ведь мы создаем активный компонент. В графе **Symbol Name Stem** вводим имя наших созданных сегментов. Вводится только общая часть имени, которая слева до первого символа подчеркивания, в нашем случае это **SEGYEL**. После этого становится доступным для ввода количество состояний – **No. of States**. Оно численно равно количеству различных видов символов – цифра после первого подчеркивания с учетом нулевого. Это означает, что если у нас номер последнего символа (в нашем случае добавленной десятичной точки) равен семи и нулевой символ – сегмент «**a**», то в этой графе мы указываем число восемь. Не ошибайтесь с подсчетом в этом месте, иначе один из символов работать не будет. Ну и, наконец, поскольку мы создаем многоэлементный индикатор, в котором каждый символ связан с состоянием определенного вывода компонента, включаем флажок **Bitwise** – «бит-зависимый» (Рис. 20).

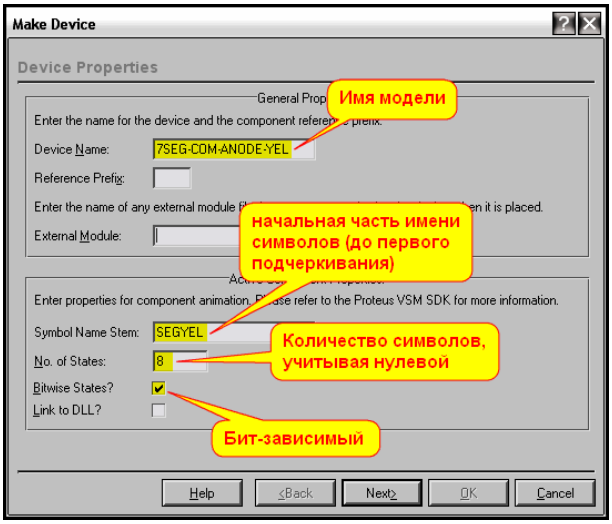


Рис. 20

В остальных вкладках функции **Make Device** можно пока ничего не заполнять и лихо проскочить их до последней, нажимая кнопку **Next**. На последней вкладке нам необходимо причислить нашу модель к какой-либо группе компонентов, либо создать новую. Я не стал мудрить в данном случае – выбрал из существующих, изменив только описание компонента – графа **Device Description**. Ну и сохранил это все пока в библиотеке **USRDVC** (Рис. 21).

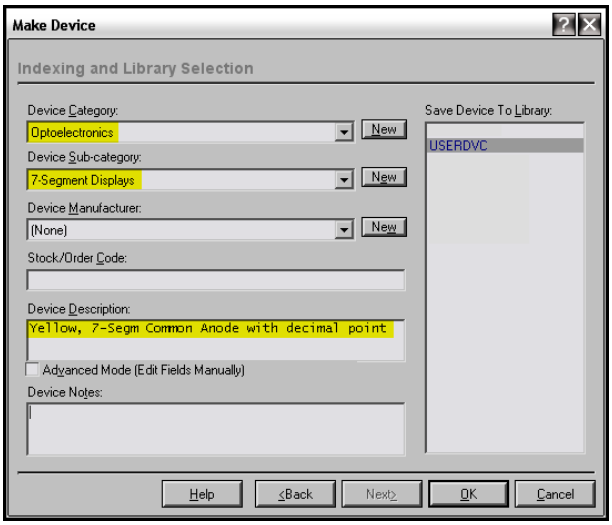


Рис. 21

Теперь наша модель присутствует в селекторе компонентов, и мы можем поместить ее в проект. Пора приделать к ней дочерний лист, чтобы на нем смоделировать внутреннюю структуру. Тут все, как и прежде – заходим в свойства и устанавливаем галочку **Attach hierarchy module**. Теперь мы можем спокойно проследовать на дочерний лист и заняться восстановлением структуры модели по

файлу **7SEGCOMK.MDF**. Если бы мы не добавили еще один элемент – десятичную точку, то можно было бы и вообще использовать этот файл, но теперь нам придется добавить недостающую часть внутренней структуры и для нее. Вообще, для активных **Bitwise** компонентов рисовать структуры – одно удовольствие. Достаточно воспроизвести структуру для одного символа и затем размножить ее через кнопку **Block Copy**. На рисунке 22 приведена часть восстановленной по **7SEGCOMK.MDF** структуры с дочернего листа нашей модели. Здесь четко просматриваются два одинаковых «канала» для сегмента «**A**» и для сегмента «**B**». Каждый канал состоит из примитивов токового пробника **RTIPROBE** и аналогового выключателя **VSWITCH**. Всего таких каналов восемь, по числу используемых символов. Все отличие состоит в номере свойства **Target Element** у токовых пробников **IP1…IP8** (на рисунке это свойство ELEMENT специально сделано видимым).

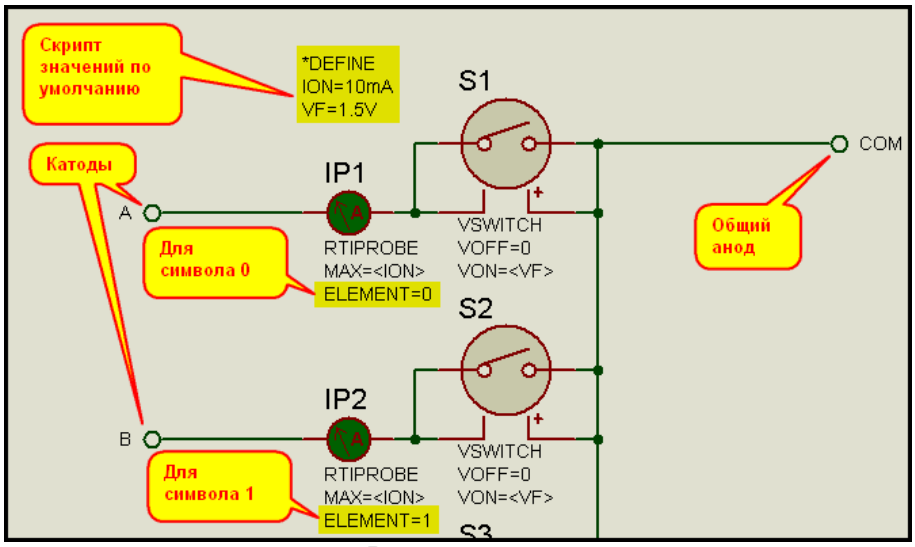


Рис. 22

Рассмотрим предназначение элементов одного канала. Аналоговый выключатель в данном включении служит пороговым элементом по напряжению с напряжением включения **VON=<VF>**. Напряжение включения **VF** задано в скрипте ***DEFINE** и равно 1.5V для всех каналов. Позже, мы пропишем его в свойствах модели с возможностью изменить из основного листа. Кроме того, в свойствах каждого **VSWITCH** в соответствии с **MDF** прототипа задано сопротивление в выключенном состоянии **Off Resistance** ROFF=100k и во включенном **On Resistance** RON=10. Таким образом, если между левыми и правыми группами выводов **VSWITCH** приложить напряжение менее полутора вольт, то он не включится и его сопротивление будет 100кОм, а если свыше 1,5В, то он перейдет во включенное состояние с сопротивлением 10 Ом между верхними по схеме выводами. Обратный переход произойдет при напряжении VOFF в нашем случае при 0В. Мы уже рассматривали управляемый ключ **VSWITCH** в **п.4.13** и больше я на нем останавливаться не буду. А вот на свойствах примитива **RTIPROBE** и его собратьев по назначению остановимся подробнее, так как они являются основой для создания активных индикаторов. Все они расположены в библиотеке **Modelling Primitives** в подпапке **Realtime (Indicators)** и назначение их служить пробниками для определения состояний точек (цепей) схемы с выводом результата в виде меняющихся графических символов. Аналоговыми индикаторами являются два из них токовый **RTIPROBE** и напряжения **RTVPROBE** (Рис. 23). Соответственно токовый включается в контролируемую цепь последовательно и индицирует ток в ней, а пробник напряжения подключается к двум контролируемым точкам и индицирует напряжение между ними. Текущее состояние (индицируемый символ) определяется по схожим в написании формулам – я привел их на рисунке над соответствующими моделями. В них:

STATE – текущее состояние (отображаемый в данный момент символ);

NUMSTATES – возможное количество состояний (символов индикации) для аналогового индикатора (вспомните восемь разных по свечению состояний светодиода на рисунке 7 из п.7.2 выше) или количество бит-зависимых символов с двумя состояниями для **Bitwise** индикатора (это то, что мы прописывали на первой вкладке **Make Device** в графе **No. of States** на рисунке 20);

CURRENT (или **VOLTAGE**) – текущее значение тока (напряжения);

MIN и **MAX** – заданные в свойствах соответственно: минимальное и максимальное значение тока (напряжения).

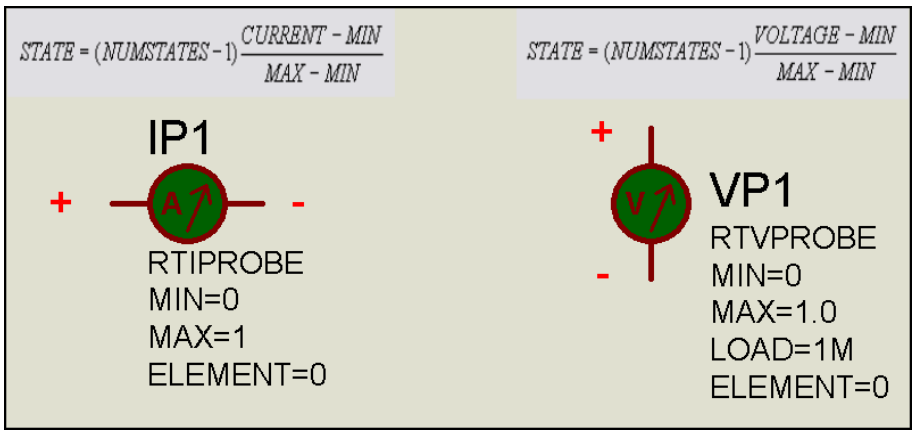


Рис. 23

Ну и сразу же рассмотрим оставшиеся свойства этих примитивов. На рисунке 24 приведено приведено окно свойств пробника напряжения – у него на одно свойство больше, чем у токового.
Load Resistance – нагрузочное сопротивление самого пробника (только для пробника напряжения).
Target Element – целевой элемент (только для **Bitwise** индикаторов). Это как раз номер того символа (сегмента), который контролируется данным пробником в нашем случае.

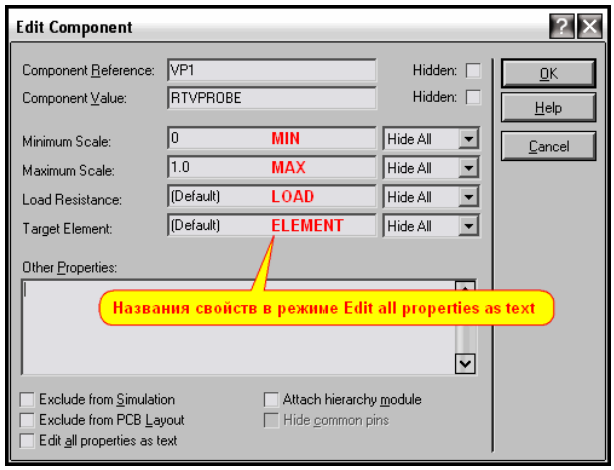


Рис. 24

Итак, в нашем случае использован токовый пробник **RTIPROBE** с установленным **MAX=<ION>**, которое в свою очередь в скрипте задано 10mA. Так как мы используем **Bitwise** режим, то соответствующий символ (сегмент) станет активным (светящимся) в том случае, если ток через пробник стане 10мА или выше, а это произойдет при срабатывании ключа **VSWITCH**. Вот собственно и весь принцип работы одного канала индикатора.
Ну и несколько слов о цифровых примитивах **RTDPROBE**. Однобитные примитивы индикаторов ведут себя предсказуемо: если на входе лог. 0, то соответствующий **Target Element** (символ) не активизирован, если на входе лог. 1, то он активизирован. В той же папке имеются многовходовые **RTDPROBE**. Детально я их не исследовал, но при беглой логика входов абсолютно непонятна – что-то похожее на исключающее ИЛИ. К сожалению и в **HELP** на них описание весьма скудное. Соответственно свойства всех рассмотренных примитивов описаны в хелпе **ProSPICE Primitives** в разделе **Real Time Modelling Primitives**. Владеющие английским языком могут прочесть в оригинале. Ну а мы вернемся к «нашим баранам», т.е. индикаторам.
Есть еще одна немаловажная «фишка», об которую я в начале освоения Протеуса набил немало шишек. При создании модели индикатора на первой вкладке **Make Device** мы не указали префикс модели – **Reference Prefix**. Я пропустил его умышленно, поскольку именно так сделано и у существующих моделей семисегментных индикаторов. Но, со всей ответственностью могу утверждать, что наша активная модель с дочерним листом без него «светиться» не будет. Будет потреблять ток, никаких ошибок и предупреждений не будет, но нормальной работы мы не добьемся, пока не применим «превентивных мер». А сделать над всего-навсего следующее. Либо задать префикс уже в свойствах готовой модели в строке **Component Reference**, либо, находясь на дочернем листе зайти в верхнем меню **Design => Edit Seet Properties** и там задать имя дочернему листу (Рис. 25). Причем имя должно быть обязательно задано только заглавными буквами латиницы или цифрами. Я и тут не стал философствовать, а просто набрал символику индикаторов **HG**.

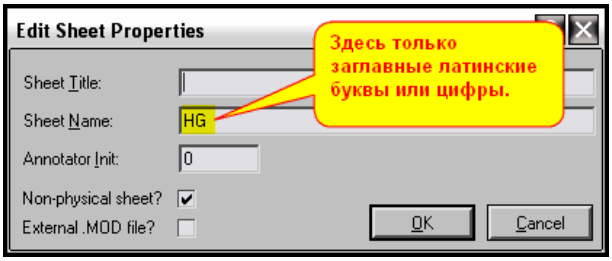


Рис. 25

Вот теперь можно вернуться на основной лист, подключить соответствующее питание и запустить симуляцию, чтобы убедиться, что наш индикатор работает (Рис.26).

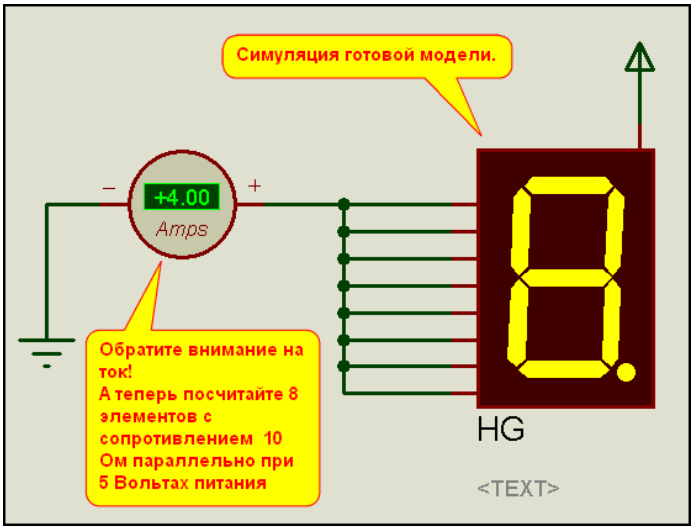


Рис. 26

Я умышленно врезал в цепь амперметр, чтобы показать, что наша модель чисто аналоговая. При питании от терминала 5V (по умолчанию) она потребляет ток 4А (!!!). Определяется он в данном случае параллельным включением всех 8 **VSWITCH** для которых задано сопротивление RON=10 Ом. Учитывайте это в своих разработках, используя подобные модели. Это не относится к индикаторам на основе DLL, о которых речь пойдет позже.

Ну и в заключение нам осталось сформировать файл MDF с дочернего листа нашего проекта, который представлен во вложении в папке **Model_with_Child**. Как это делается вы уже должны знать наизусть.

Окончательный вариант с отключенным дочерним листом представлен в папке вложения **Final_model_with_MDF**. Там же лежит и скомпилированный файл **7SEG_POINT_COM_AN.MDF**. Теперь нам осталось еще раз пройти процедуру Make Device для модели и на третьей вкладке добавить сам файл **MDF** (Рис. 27), а также через **Blank Item** по аналогии с прототипом два свойства **ION** и **VF**, которые были у нас в скрипте ***DEFINE** дочернего листа (Рис 28 и 29 соответственно). Не забудьте открепить сам дочерний лист – снять галочку.

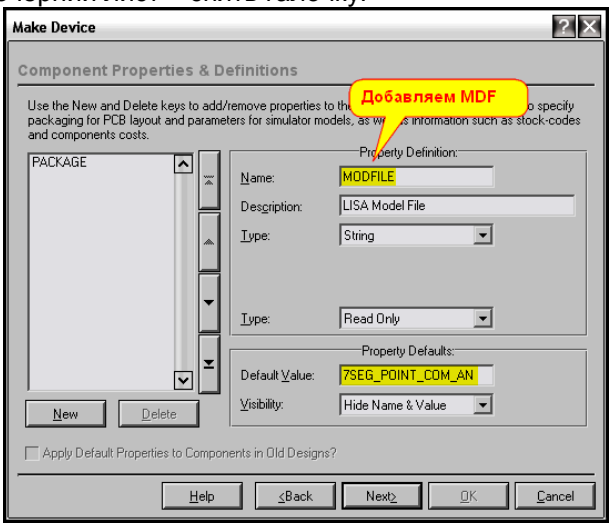


Рис. 27

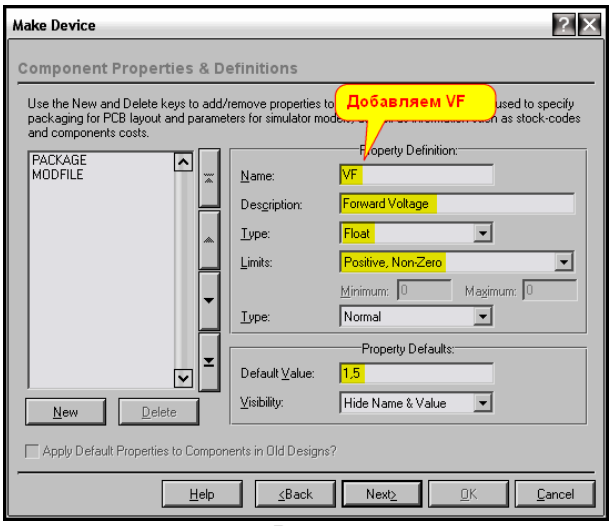


Рис. 28

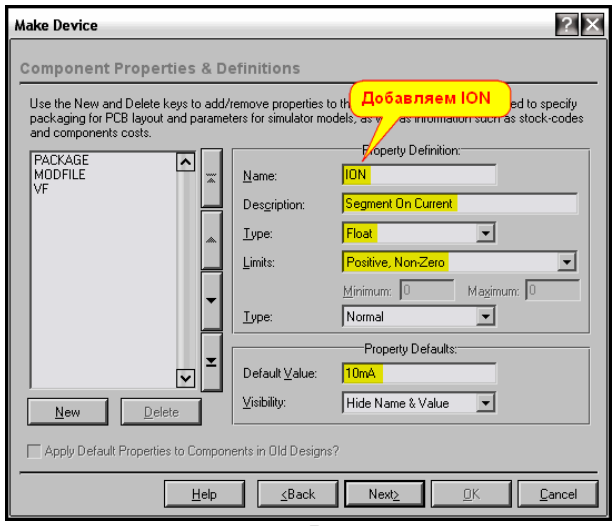


Рис. 29

На этом модель полностью готова. Желающие могут перенести ее из **USRDVC** в какую-либо другую библиотеку, а файл MDF поместить в папку **MODELS** Протеуса для дальнейшего использования. Далее можете самостоятельно попрактиковаться в «раскрашивании» семисегментных индикаторов. Кстати, если использовать символы, которые мы создали для желтого и только менять цвета, то файл **MDF** для всех будет один и тот же, т.е. процедуру «приклеивания» дочернего листа повторять уже не придется. Мы просто перекрашиваем сегменты, создаем новый набор символов нужного цвета, затем создаем модель с оригинальным именем и присоединенным набором сегментов нужного цвета, а далее на третьей вкладке добавляем наш MDF и дополнительные два свойства.

Можно также самостоятельно пораскрашивать светодиоды – там и MDF менять не надо, оставляете родной. Я же на этом временно закончу рассмотрение индикаторов и перехожу к теме создания моделей на основе существуюших **DLL**, т.е. приступаем к созданию ранее представленного и обещанного регулируемого источника питания.

К содержанию

8. Активные модели на основе существующих DLL.

8.1. Немного «тумана» о DLL и о том, что будет, а чего не будет в этом разделе.

Вот мы и подошли к тому моменту, когда количество изложенного материала даст нам и качественный скачок. Но сначала о том, что же такое **DLL**. **Dynamic-link library** – библиотека динамической компоновки – так гласит Википедия. Концепция применения **DLL** изначально заключалась в том, чтобы создать исполняемые модули, которые могли бы независимо использоваться различными приложениями. Да и структура динамических библиотек во многом схожа со структурой исполняемых файлов, они содержат исполняемый код, данные и ресурсы, которые могут полностью или частично использоваться разными приложениями Windows. Характерным примером DLL являются драйвера устройств. Компоновка программного кода в динамические библиотеки возможна в большинстве программных сред, работающих с языками высокого уровня. Возможность использования единожды загруженной в память DLL различными процессами в динамическом режиме позволяет добиться гибкости и высоких скоростей обработки информации.

Протеус, как программный продукт, работающий под управлением MS Windows, не является исключением. Именно использование DLL позволяет проводить имитацию поведения микроконтроллеров и других сложных электронных компонентов в режиме реального времени. Подавляющее количество библиотек для программных моделей в **ISIS** написано самим разработчиком – фирмой **Labcenter Electronics** в среде **MS Visual C++** и скомпоновано в динамические библиотеки с именем имитируемой модели или группы и расширением DLL. В ранних версиях Протеуса, до версии 6.2 включительно в состав установочного пакета входили и средства для разработки программных моделей - **Proteus VSM SDK**. Для «неанглоязычной» и абсолютно незнакомой с программированием публики привожу расшифровку аббревиатуры и перевод. **VSM** – Virtual System Modeling – виртуальное системное моделирование, **SDK** – Software Development Kit – комплект средств разработки. Теперь, надеюсь, понятно, о чем идет речь. Это заголовочные файлы для среды C++ с расширением **CPP**, располагавшиеся в папке **INCLUDE**, а также файл помощи по их использованию. Кстати, ссылка на этот файл до сих пор фигурирует в меню Help ISIS, только вместо реального файла помощи там торчит «заглушка». Как это не печально, но, начиная с версии 6.3, доступ к средствам разработки был закрыт, поскольку фирма опасается плагиата со

стороны возможных конкурентов. Если кто-то чувствует себя вполне уверенно в программировании на **C++**, причем не элементарном – две три функции, а с использованием классов, то может поискать старые версии Протеуса, к которым прикладывались средства для разработки. При тщательном поиске в сети можно обнаружить и PDF-вариант HELP, называется он **Proteus VSM DSK** (именно DSK а не SDK – не знаю кто, но допустил «очепятку» и так и не поправил ее).
Я же в последующем материале не ставлю перед собой цель обучить вас программированию на **C++**, и переписыванием вышеупомянутого VSM DSK на русском языке тоже увлекаться не собираюсь. Моя цель гораздо проще. Я хочу показать в этом разделе, что умело используя уже существующие **DLL**, можно создать достаточно «продвинутые» модели для применения в своих проектах. И для этого совсем не надо быть корифеем в **C++**, можно даже совсем не знать этого языка. Достаточно проявить нашу русскую смекалку и сообразительность, ну и немного «упертости» в достижении результата. Приступим…

К содержанию

8.2. Библиотека SETPOINT.DLL и задаваемые для нее параметры на примере температурного датчика LM20.

Само название библиотеки, если разложить его на составляющие и перевести, говорит о ее назначении, **Set point** в переводе с английского – «установить точку». Именно для этих целей и написана **SETPOINT.DLL**. С параметрами, которые можно использовать при использовании данной библиотеки, проще всего познакомиться на конкретном примере. **SETPOINT.DLL** достаточно широко применяется в моделях Протеуса и мы встретим ее почти везде, где встречаются маркеры **INCREMENT** и **DECREMENT** и «миниатюрный дисплей» для вывода числового значения изменяемого параметра (Рис. 30). Для изучения свойств я выбрал модель температурного датчика LM20, хотя можно было воспользоваться и любой другой моделью с присутствующими в ее составе характерными элементами. Вы уже наверняка применяли в своих проектах подобные модели и знаете, что клик мышкой по маркеру INCREMENT (стрелка вверх) дает приращение параметра на один шаг (в этой модели **Temperature Step** в свойствах), а клик по маркеру **DECREMENT** (стрелка вниз) уменьшает значение на один шаг.

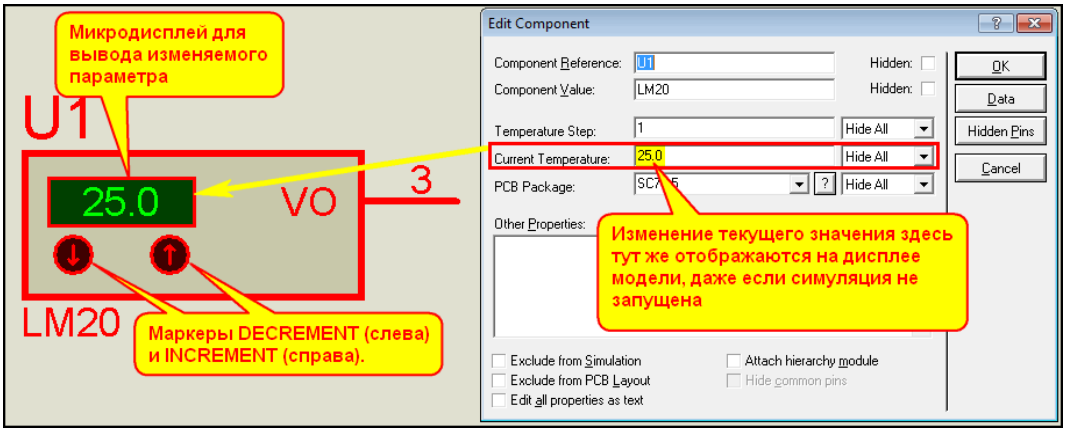


Рис. 30

Чтобы познакомиться с тем, как подключена библиотека **SETPOINT.DLL** к модели LM20, а заодно и узнать об остальных параметрах воспользуемся все той же, любимой мною, функцией **Make Device**. Выделяем LM20 в поле проекта, задаем **Make Device** и смотрим что же интересного на первой вкладке (Рис.31).

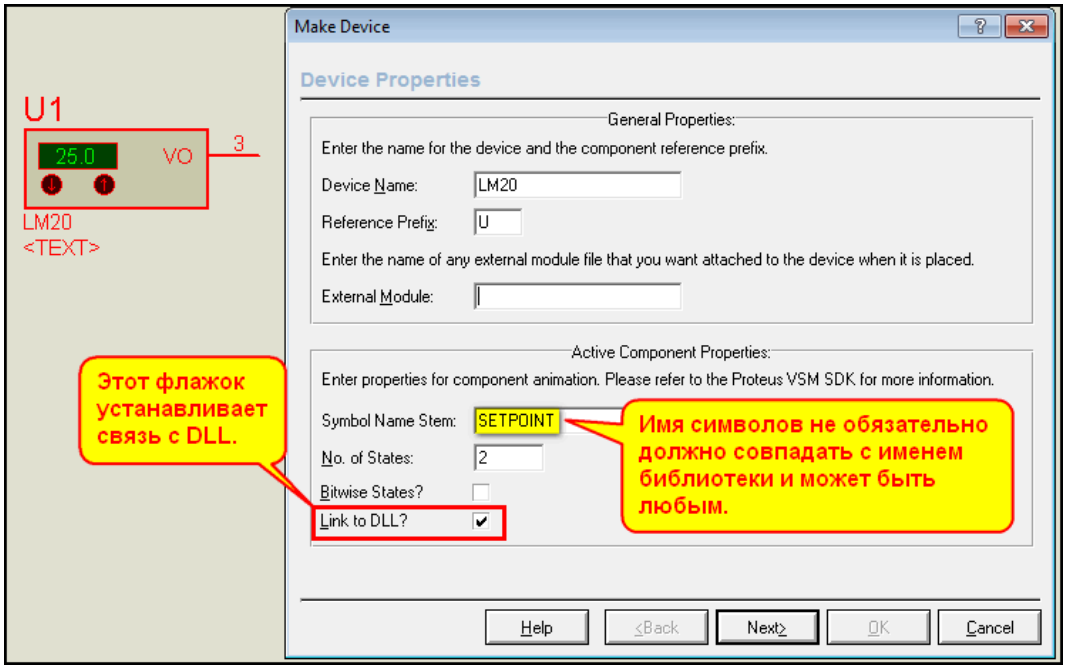


Рис. 31

Здесь мы можем убедиться, что наша модель является активной, поскольку заполнена нижняя часть вкладки – **Active Component Properties**. В составе модели имеются два символа с именем **SETPOINT**. В данном случае имя символов совпадает с именем библиотеки, но это совсем не обязательное условие работоспособности модели. Просто здесь имеет место случайное совпадение. А вот самым нижним флажком мы еще не пользовались. Именно он и привязывает активные свойства модели к программной библиотеке **DLL**. **Link to DLL** так и переводится на русский – «связь с DLL». Ну, к символам активной модели мы вернемся позже, а сейчас, чтобы лишний раз не выпадать из **Make Device**, проследуем на третью вкладку и посмотрим что еще интересного в свойствах нашей модели. И в первом же свойстве **MODDLL** мы встречаем привязку уже конкретно к библиотеке **SETPOINT.DLL** (Рис.32). Более, кроме того, что этому свойству присвоен тип **Hidden** (скрытое) нас в этом свойстве ничего не заинтересует.

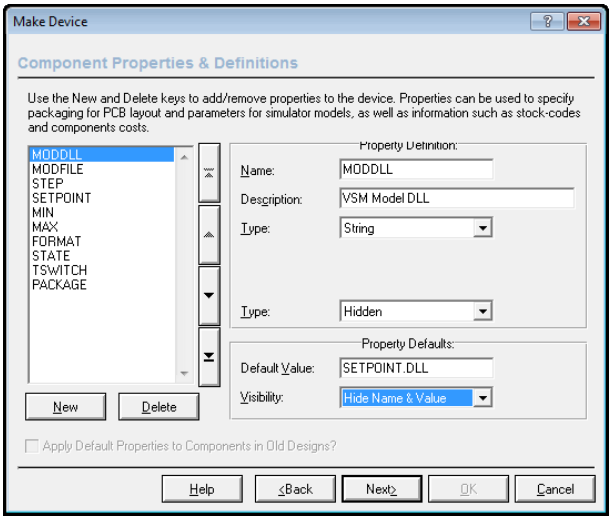


Рис. 32

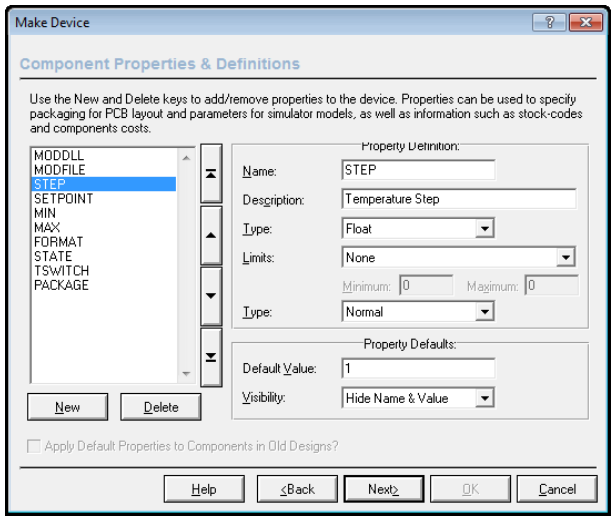


Рис. 33

Свойство **MODFILE** относится к конкретному **MDF** модели, и мы на нем пока останавливаться не будем, а рассмотрим только те, что связаны с **SETPOINT.DLL**.

STEP – шаг (Рис.33). Для этого свойства и последующих, относящихся к **SETPOINT**, при добавлении их к создаваемой модели следует выбирать опцию **New=> Blank Item** и вводить имя вручную, поскольку в стандартных они не прописаны. Написания имен должны быть именно такими, без всяких там «загогулин» и грамматических ошибок, т.к. именно так они фигурируют в **SETPOINT.DLL**. Итак, шаг определяет единичное изменение регулируемого параметра при нажатии на маркер **INCREMENT** или **DECREMENT**. В графе **Description** для этого свойства вы вольны писать все, что заблагорассудится, в том числе и на кириллице. Поскольку в данный момент мы рассматриваем датчик температуры там и написан **Temperature Step**. **Type** определен как **Float**, т.е. можно использовать дробные значения. Ограничения – **Limits** не установлены. В следующей

графе **Type** определено будет ли показан данный параметр при открытии окна свойств модели и доступен ли он для редактирования. Значение **Normal** в данной графе предусматривает и то и другое. Ну и, наконец, в **Property Defaults** назначено значение по умолчанию **1** и для **Visibility** определено **Hide Name & Value**, т.е. по умолчанию рядом с моделью на месте серого **<TEXT>** это свойство демонстрироваться не будет.

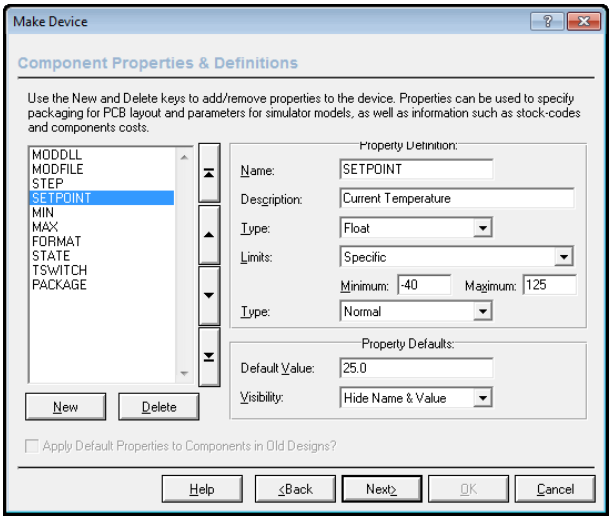


Рис. 34

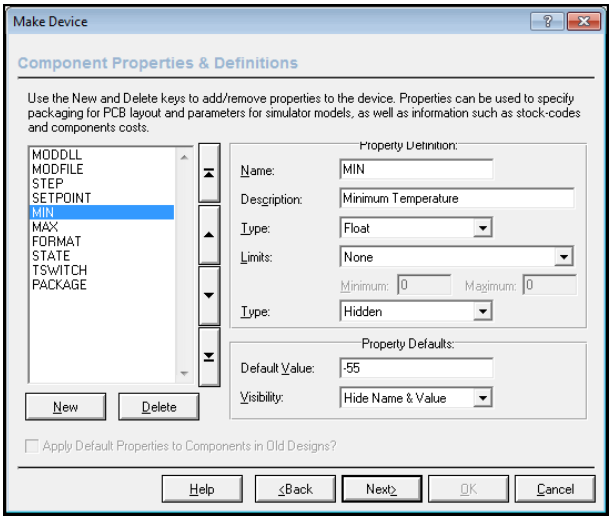


Рис. 35

SETPOINT – установленная точка (Рис.34). Это свойство задает модели начальное (стартовое) значение. Опять-таки, поскольку у нас датчик температуры, то ему и задано **Description** как **Current Temperature**. Тип задан тоже **Float**, а вот на графу **Limits** я прошу обратить особое внимание. В ней из раскрывающегося списка выбран тип Specific (надеюсь, хоть это слово переводить не надо) и ниже заданы особые ограничения минимальное **-40** и максимальное **125**. Т.е. это предельные значения для задания температуры. Я не знаю, какие цели преследовал автор модели, задавая именно эти значения здесь, так как для большинства модификаций датчиков **LM20** эти значения по даташиту равны соответственно **-55** и **130**. Но в одном я ему благодарен, поскольку у меня есть возможность показать, как это ограничение работает в **ISIS**. Если вы попробуете открыть окно свойств датчика (Рис. 30) и ввести там для **Current Temperature**, допустим значение **128** и попытаетесь потом закрыть окно свойств, то получите предупреждение о недопустимом значении температуры. Аналогичное сообщение выскочит и при задании температуры ниже **-40**. Ну, больше для этого свойства ничего интересного нет, следующий **Type** задан **Normal**, поэтому в окне свойств модели (Рис. 30) мы видим **Current Temperature** отдельной строкой с доступным для изменения окошком значения, которое по умолчанию для «свежевытащенной» из библиотеки модели равно **25** градусам (окошко **Default Value** на рисунке 34).

Ну а теперь небольшой фокус. Для отдельно стоящих свойств **MIN** (Рис. 35) и **MAX** (Рис. 36) заданы соответственно именно те значения из даташита **-55** и **130**. Эти свойства передают программной модели **SETPOINT** пределы изменения нашего параметра с помощью кнопок-маркеров **INCREMENT** и **DECREMENT**. Так вот, автор здесь задал значения шире, чем те пределы в свойстве **SETPOINT**. И с помощью этих кнопок мы можем изменять температуру именно в пределах **-55…130**, и Протеус при этом ругаться не будет. Вот такой парадокс. Ну, раз уж мы плавно перескочили к рассмотрению свойств **MIN** и **MAX**, то хочу обратить ваше внимание на то, что для них нижний **Type** задан **Hidden** (скрытый). Поэтому в окне свойств датчика **LM20** (Рис. 30) мы их не видим, но можем увидеть и даже менять заданные значения, если поставим галочку **Edit all properties as text** в левом нижнем углу этого окна. Более о них сказать нечего, **Minimum Temperature** и **Maximum Temperature** говорят сами за себя.

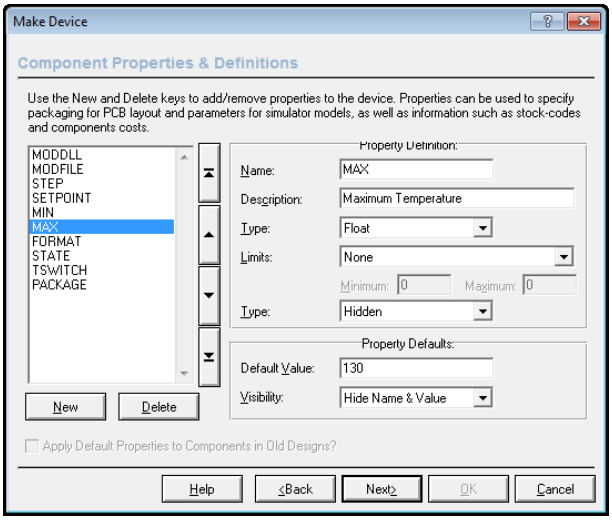


Рис. 36

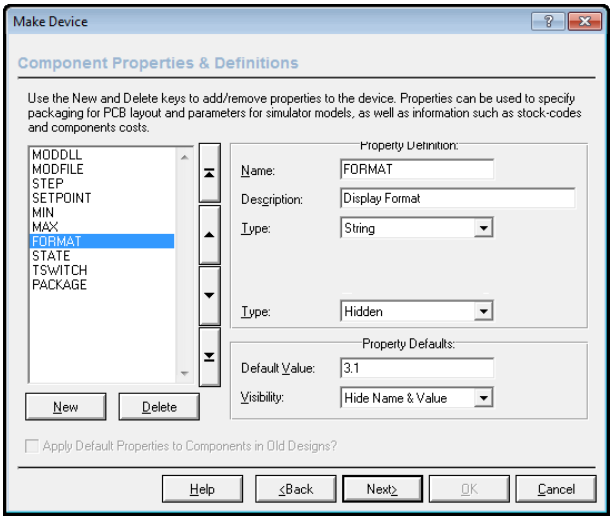


Рис. 37

FORMAT – название этого свойства тоже не нуждается в переводе, а относится оно к формату числа, выводимому в окошечко зеленого дисплея. Это подтверждается и тем, что в **Description** автор модели указал **Display Format** (Рис. 37). Обратите внимание, что верхний **Type** для этого свойства задан, как **String** – текстовая строка. Нижний **Type** тоже скрытый, поэтому оно видно только при установленном флажке **Edit all properties as text**. Ну а теперь о самом главном для этого свойства – **Default Value**. По умолчанию оно задано как **3.1**, что означает три знака до десятичной запятой и один после. При желании эти значения можно изменять, только следите за тем, чтобы общее число символов уместилось в зеленом окошке.

Следующие два свойства **STATE** – активное состояние при старте симуляции и **TSWITCH** – время переключения для большинства активных моделей на основе **SETPOINT** в основном будут иметь именно те значения, которые показаны на рисунках 36 и 37 соответственно. Первое из них отвечает за то, какой параметр будет активным (доступным для изменения) при старте симуляции. Поскольку он у нас всего один, то и **STATE=0**. Ну а время переключения **TSWITCH=1ms** – это то время, за которое наше значение изменится после нажатия на соответствующую кнопку-стрелку. Соответственно им тоже задан нижний **Type** скрытый, а в ряде случаев ими, в частности **TSWITCH**, можно и вообще пренебречь и при создании своих активных моделей не указывать.

Ну а мы на этом закончим нудную, но нужную для понимания того, что мы будем делать ниже описательную часть, и переходим к практическим действиям – созданию активных моделей на основе **SETPOINT.DLL**.

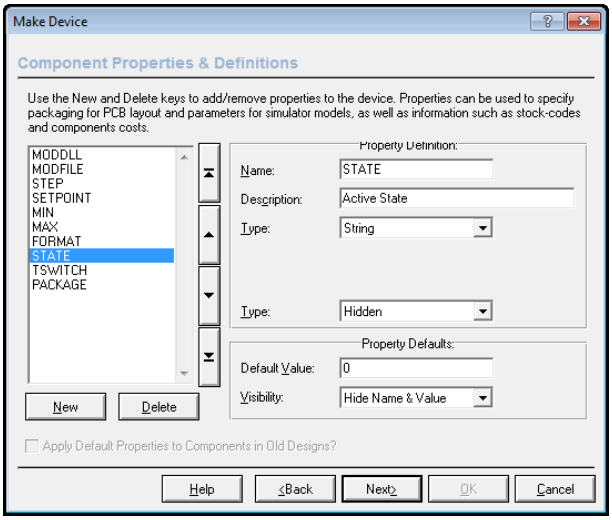


Рис. 36

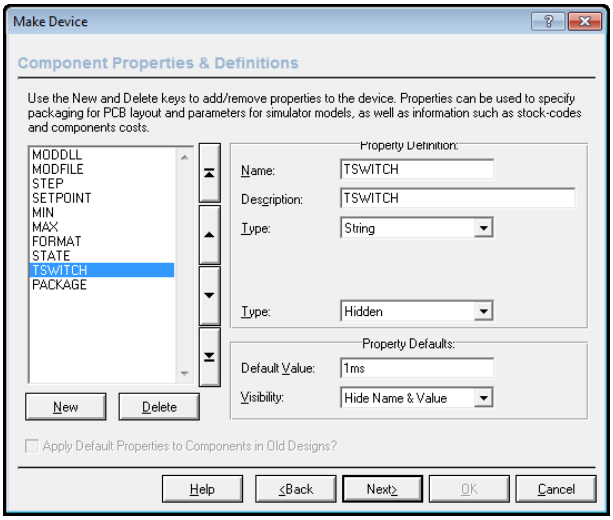


Рис. 37

К содержанию

8.3. **Простые регулируемые источники на основе** SETPOINT.DLL.

Отлаживая свои проекты в среде Протеуса, как то я обратил внимание на некоторое упущение разработчиков программы, доставляющее массу неудобств. В **ISIS** абсолютно отсутствуют готовые средства, с помощью которых можно в процессе выполнения симуляции изменять напряжение или ток, в какой либо цепи. Конечно, можно их реализовать с помощью обычных генераторов и тех же активных моделей потенциометров, но это получается настолько громоздко и неудобно в использовании, что весь эффект применения сводится на нет. И вот однажды, отлаживая

очередную схему, в которой присутствовала активная модель датчика температуры, я подумал – а нельзя ли подобным образом реализовать, например, изменяемый источник напряжения для питания какой либо части схемы. И удобно, и наглядно. Ну и дальше все просто, «подумано» – сделано.
Я изучил, как сделана модель датчика температуры, что мы проделали выше, а затем заглянул в ее файл **MDF**. Для любопытствующих я поместил уже извлеченный из **NATDAC.LML** файл **LM20.MDF**. И тут меня ждал весьма неприятный сюрприз. В **PARTLIST** файла модели (см. ниже) фигурировал элемент **VOUT**, который отсутствует в библиотеках Протеуса. Пошарив в других **MDF**, использующих **SETPOINT.DLL**, я обнаружил его во всех списках компонентов этих моделей.

```
*PARTLIST,6
AVS1,AVS,"(-3.88E-006*V(A,B)^2)+(-1.15E-002*V(A,B))+1.8639",PRIMITIVE=ANALOGUE
C1,CAPACITOR,1uF,PRIMITIVE=ANALOGUE
I1,CSOURCE,10uA,PRIMITIVE=ANALOGUE
R1,RESISTOR,10k,PRIMITIVE=ANALOGUE
R2,RESISTOR,100,PRIMITIVE=ANALOGUE

V1,VOUT,1V,MODDLL=SETPOINT,PRIMITIVE=ANALOG,SETPOINT=<SETPOINT>
```

Поначалу меня такой сюрприз обескуражил, но логически помыслив, я пришел к выводу, что раз он так широко используется, то, скорее всего, реализован в виде встроенного в среду примитива и можно попробовать «обмануть» Протеус, подсунув ему собственную модель с теми же свойствами. Попробуем реализовать его на основе двухполюсника генератора напряжения **VSOURCE** из библиотеки **Simulator Primitives**. Функционал – источник постоянного напряжения у них совпадает, значит осталось только придать прототипу недостающих свойств. Помещаем извлеченный из библиотеки **VSOURCE** в поле проекта и применяем к нему мою излюбленную **Make Device**. Смотрим на последнюю строчку приведенного выше **PARTLIST**. Наша модель должна иметь имя **VOUT** и префикс **V**, что и прописываем на первой вкладке (Рис. 40). Напомню, что в ней все свойства разделены запятыми, а **VALUE** стоит третьим сразу после имени **VOUT**.

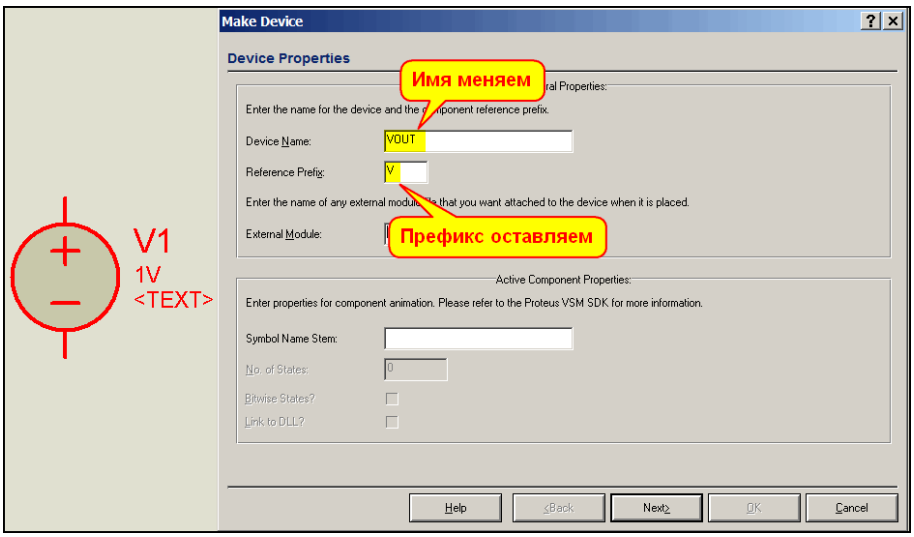


Рис. 40

Вторую вкладку проходим транзитом, поскольку корпус нашему девайсу не нужен и переходим на третью. Здесь убеждаемся, что уже присутствующие свойства **PRIMITIVE** (Рис 41) и **VALUE** (Рис. 42) соответствуют записанным в строке для **VOUT** из приведенного выше **PARTLIST**.

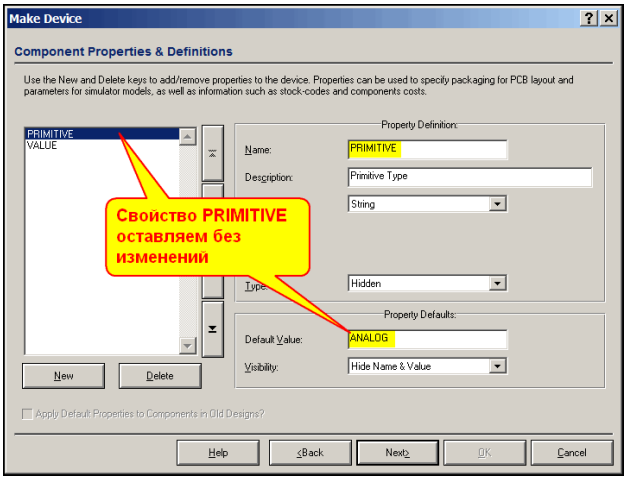


Рис. 41

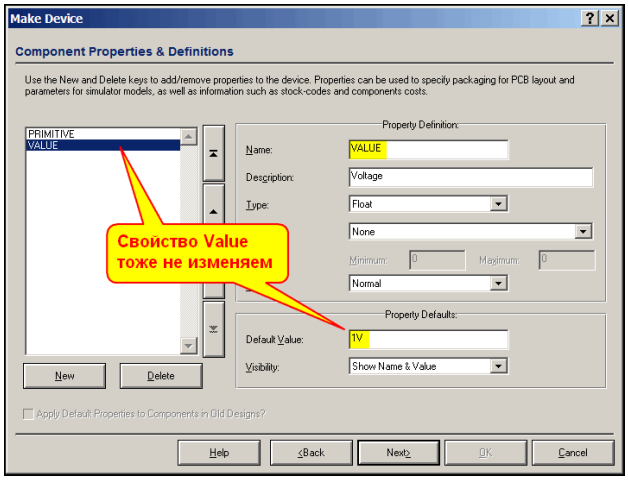


Рис. 42

Теперь добиваем через кнопку **New** недостающие **MODDLL** (Рис. 43) и **SETPOINT** (Рис. 44). Последнее, по большому счету, добивать необязательно, всегда можно добавить вручную в окне свойств компонента, но на первых порах, чтобы не забыть о его существовании - лучше вставить.

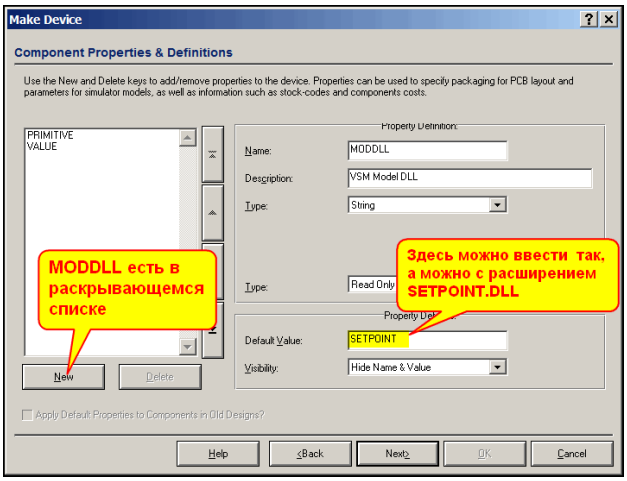


Рис. 43

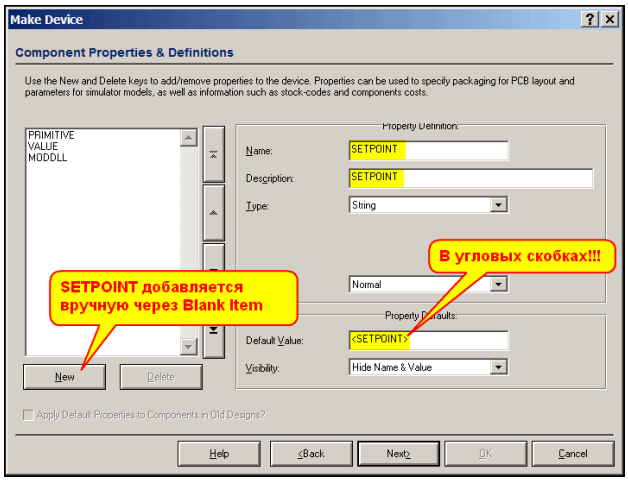


Рис. 44

Обратите внимание, что его **Default Value** стоит в угловых скобках (знаки больше-меньше на клавиатуре). Это дает команду Протеусу подставлять значение из внешнего по отношению к модели окружения. После этого доходим до последней вкладки и сохраняем наш VOUT в какой-либо библиотеке. Я тут не мудрствовал и сохранил в **USRDVC**, оставив все категории компонента, как и у **VSOURCE**, чтобы не забыть, где его впоследствии искать (Рис. 45).

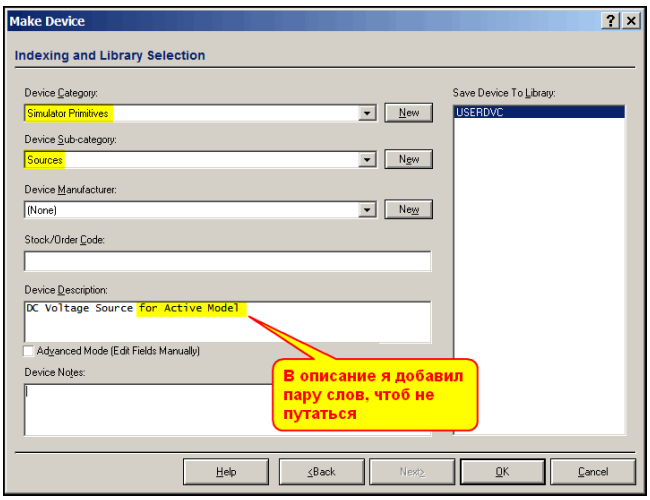


Рис. 45

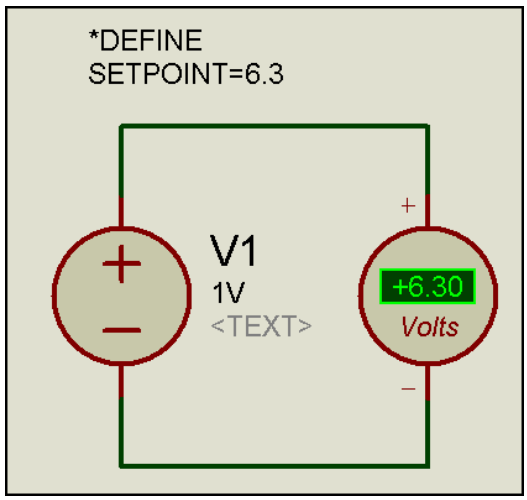


Рис. 46

Если теперь вытащить наш **VOUT** в поле проекта и запустить симуляцию, то вылезет желтое предупреждение в логе об отсутствии значения для **SETPOINT**, т.е. **ISIS** не нашел конкретного значения. Однако если поместить в поле проекта скрипт следующего содержания:

```
*DEFINE
SETPOINT=6.3
```

Наша модель начинает благополучно работать и «выдавать на гора» то напряжение, которое задано в скрипте. Такая подстановка вам уже знакома из создания схематичных моделей, но я счел нужным лишний раз напомнить – как это работает. В папке **Create_VOUT** присутствует проект, с которого сделан скриншот рисунка 46. Если кому то очень лень (некогда) заниматься такими пустяками, как создание **VOUT** – можете просто применить к стоящей в этом проекте модели **Make Device**, пройти ничего не меняя все вкладки и сохранить в собственной библиотеке Протеуса. Если ничего не трогать, то сохранится он в **USRDVC**, а в библиотеке его можно будет найти в **Simulator Primitives => Sources.**
Ну вот, теперь нам осталось создать активную графику для нашего будущего источника. Тут тоже не стоит мудрить – давайте «разберем на запчасти» все тот же **LM20** и воспользуемся его символами. Итак, помещаем в поле проекта **LM20**, **Decompose** его и видим, что у нас в селекторе символов появились два символа с именем **SETPOINT**: SETPOINT_0 и SETPOINT_1.

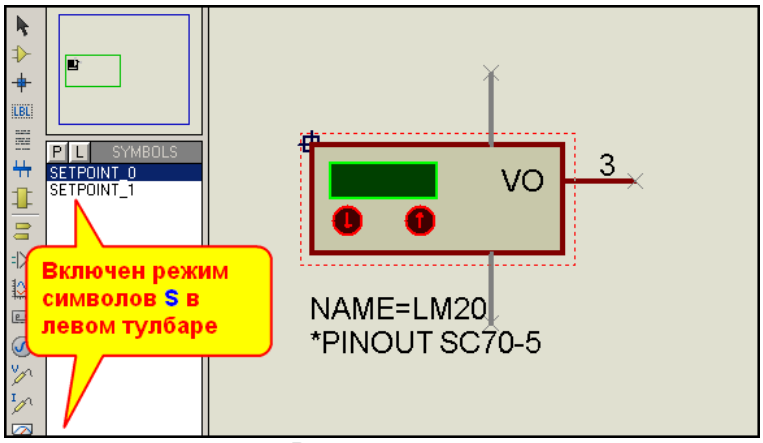


Рис. 47

Я не стану в этот раз заморачиваться с изменением символов, поскольку в активной модели, как я уже указывал, сдвиг графики относительно маркера **ORIGIN** повлечет за собой коррекцию всех остальных графических элементов. Поэтому, оставляем все как есть. На рисунке 48 приведены препарированные с помощью **Decompose** символы **SETPOINT_0** и **SETPOINT_1**, чтобы вы имели представление о местоположении маркера **ORIGIN**. Полностью разобранный на запчасти **LM20** в проекте **Prepare_LM20.DSN** папки **Create_VREG** вложения. Это как бы первый, подготовительный этап создания нашего источника. А мы переходим ко второму и заключительному.

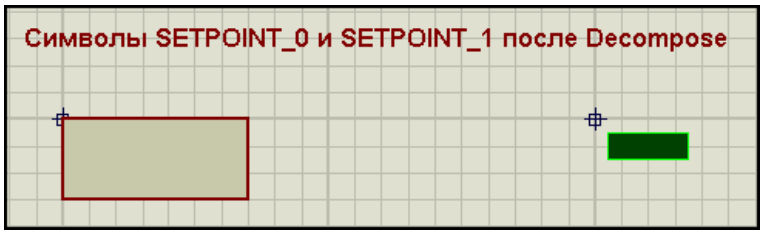


Рис. 48

Теперь нам необходимо убрать все лишнее из графики и добавить свое. Лишним в данном случае является скрипт и выводы компонента, в том числе и скрытые выводы питания, которые появились после разборки модели на запчасти. В результате у меня получилась «конструкция», показанная на рисунке 49 слева с которой я и начинаю выполнять создание нашего источника с помощью **Make Device**. На первой вкладке заполняем все так, как на рис. 49.

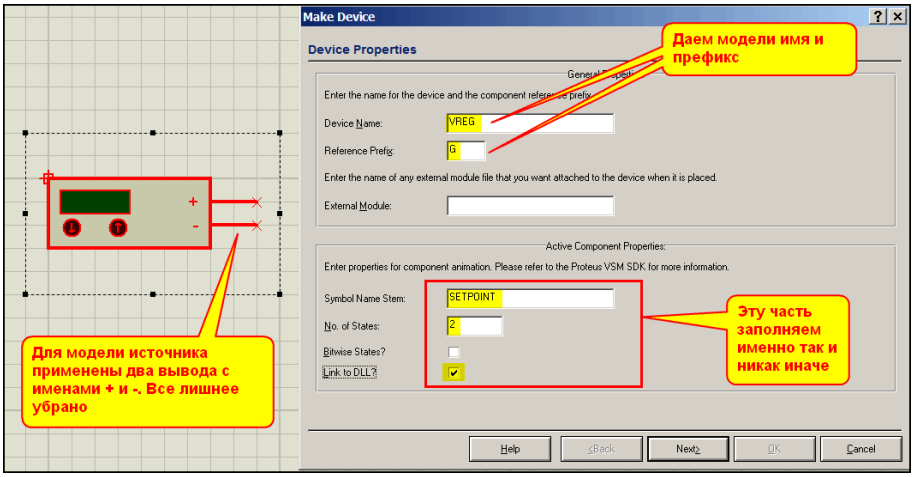


Рис. 49

Особое внимание правильности заполнения нижней части, относящейся к активным свойствам. Не забудьте, что символы **SETPOINT** должны быть доступны в селекторе **SIMBOLS** этого проекта. Я сделал так, вот тот предыдущий проект **Prepare_LM20.DSN** сохранил в этой же папке с новым именем **Create_VREG.DSN**, а после этого убрал все лишнее и добавил свои выводы (Pins). При этом символы остались доступными. Далее следуем в **Make Device** на третью вкладку и начинаем добавлять свойства по аналогии с **LM20**. У нас и так в этом параграфе переизбыток картинок, поэтому ограничусь моментом добавления **MODDLL**, при этом у нас окно свойств еще девственно чистое (Рис. 50). Не забудьте, что по умолчанию нижний **Type** для **MODDLL** будет предложен как **Read Only** (только для чтения). Это означает, что в окне свойств модели он будет показан, но не будет доступен для изменения (закрашен серым). Мне на это любоваться неохота, тем более что необходимо будет другие свойства сделать доступными, в отличие от **LM20**, поэтому здесь я выберу **Hidden** (скрытое).

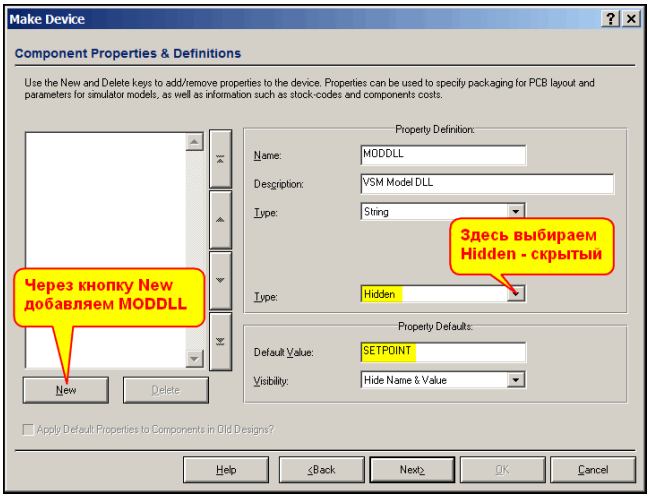


Рис. 50

Также, через кнопку **New** и опцию **Blank Item** обязательно добавляем следующие свойства:
STEP – в графе Description я набрал **Voltage Step** (Шаг напряжения), верхний **Type** – **Float**, нижний **Type** – **Normal**, **Default Value** – **0.1**.
SETPOINT – в графе Description я набрал **Current Voltage** (текущее значение напряжения), верхний **Type** – **Float**, нижний **Type** – **Normal**, **Default Value** – **0.0**. Обратите внимание, что ни здесь, нигде более я не воспользуюсь **Limits**, поскольку моя модель для отладки, но MAX и MIN заведомо достаточно большие я все же введу, иначе изменение значения с помощью кнопок будет криво работать.
MAX – в графе Description я набрал **Maximum Voltage** (Максимальное значение напряжения), верхний **Type** – **Float**, нижний **Type** – **Normal**, **Default Value** – **500**.
MIN – в графе Description я набрал **Maximum Voltage** (Минимальное значение напряжения), верхний **Type** – **Float**, нижний **Type** – **Normal**, **Default Value** – **-500**.
Я сделал максимальное и минимальное напряжения видимыми в свойствах, потому что вдруг да не хватит значения 500V, всегда можно поправить. Остальные свойства я пока вводить не буду, чтобы доказать, что и без них мы получим вполне работоспособную модель.

Доходим до последней вкладки и сохраняем нашу модель. Я сохранил ее в **Debugging Tools**, создав там дополнительную подкатегорию **Sources** (Рис. 51). Если кому то это не нравится, может сохранить в другом месте.

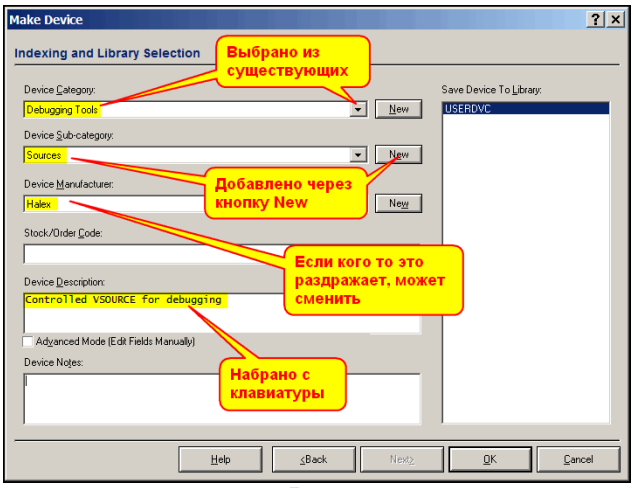


Рис. 51

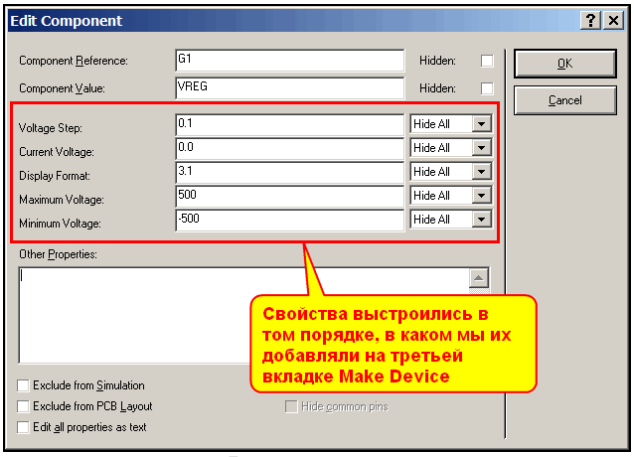


Рис. 52

Если теперь поместить нашу модель, появившуюся в селекторе компонентов в поле проекта и зайти в ее свойства, то мы увидим все наши добавки с третьей вкладки **Make Device**, причем в том порядке, в каком мы их добавляли там (Рис. 52). Обратите внимание, что **MODDLL** здесь нет, потому что для этого свойства мы выбрали нижний **Type** – **Hidden**. Порядок отображения свойств в этом окошке можно изменить. Если мы запустим еще раз **Make Device**, то на третьей вкладке можем переставить порядок расположения уже существующих свойств с помощью кнопок со стрелками справа от этого окна. Кликаем по свойству, которое надо передвинуть, и этими кнопками передвигаем его на нужное место. Средние стрелки передвигают на одну позицию, крайние верхняя и нижняя соответственно в начало или в конец. Естественно, чтобы изменения сохранились надо пройти **Make Device** до конца и подтвердить изменения при сохранении модели.
Теперь, как обычно, в свойствах устанавливаем флажок **Attach Hierarchy Module**, уходим на дочерний лист и там собираем «архисложную» субсхему нашего девайса (Рис. 53).

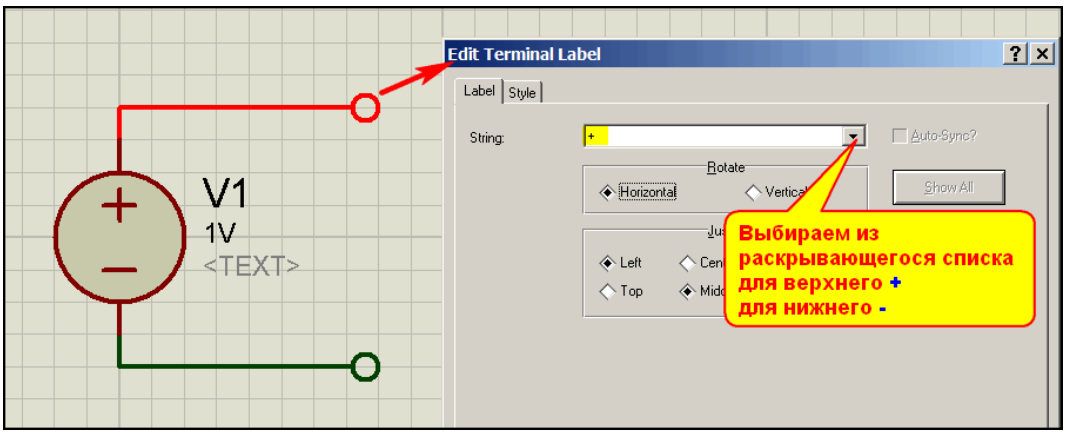


Рис. 53

Как видите, вся схема состоит из свежевыпеченного нами выше **VOUT** и двух терминалов, имена которых выбираются из раскрывающегося списка. А в списке будут фигурировать только наши два вывода модели с именами **+** и **-**. Ну вот и весь процесс, осталось вернуться на основной лист проекта, прицепить для контроля к выводам нашего источника вольтметр и запустить симуляцию. Пробуем кнопками менять напряжение и убеждаемся, что все работает как надо (Рис. 54).

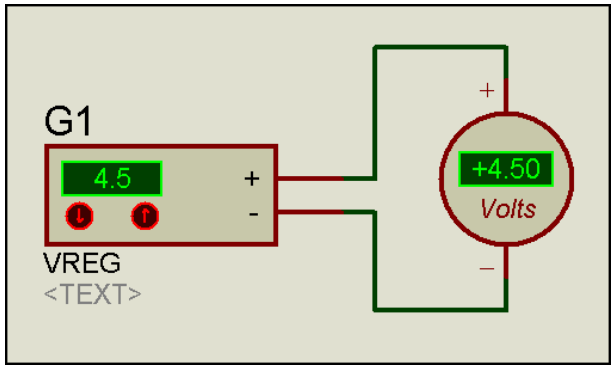


Рис. 54

Нам осталось скомпилировать с дочернего листа файл **MDF**, я назвал его **VREG.MDF**, затем запустить для нашего источника еще раз **Make Device** и на третьей вкладке добавить свойство **MODFILE**, указав ему в графе **Default Value** имя нашего скомпилированного файла. Мы это уже неоднократно проделывали при создании схематичных моделей, и, надеюсь, тут разъяснять дополнительно ничего не требуется. Окончательный вариант с файлом **MDF** представлен в папке **VREG_with_MDF** вложения.

Ну а мы теперь модифицируем наш источник и превратим его в источник тока. Модификация заключается в том, что на дочернем листе нам придется добавить еще один примитив – управляемый напряжением источник тока **VCCS** или **AVCCS** с коэффициентом передачи **1.0** (Рис. 55). После этого компилируем с дочернего листа новый файл **IREG.MDF**. Сама графика остается без изменений, но необходимо вновь пройти **Make Device**. На первой вкладке даем нашей модели другое имя, например, **IREG**. На третьей придется поправить **Description** у большинства свойств, заменив там слово **Voltage** (напряжение) на **Current** (ток), чтобы потом самим не путаться. Ну, можно еще сменить ограничения для **MIN** и **MAX** на разумные, поскольку ток в 500А нам на фиг не нужен. Ну и конечно, для **MODFILE** надо указать новое значение **IREG.MDF**. Проект с дочерним листом представлен в папке **Create_IREG** вложения. А в папке **IREG_with_MDF** уже модель с присоединенным **MDF** и сам **IREG.MDF**.

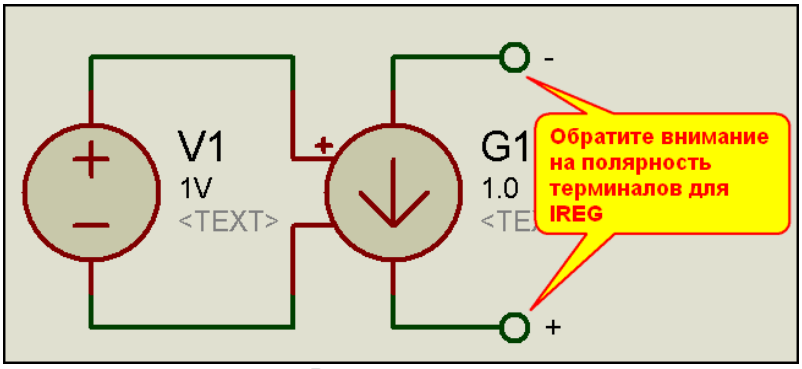


Рис. 55

Ну и несколько заключительных слов по использованию этих моделей. Я пошел навстречу «идолопоклонникам» русского интерфейса и в папке **RUS** вложения лежат проекты, в которых **Description** моделей сделан кириллицей. Для них в конце имен я добавил **RUS**. Причем файлы MDF для них остаются неизменными. Меняется только **Description** на третьей вкладке **Make Device**. Вам остается только выбрать то, что вы хотите использовать в своих проектах, протолкнуть для выбранных моделей **Make Device**, ничего не меняя во вкладках, чтобы эти модели появились в ваших библиотеках **ISIS** и переложить файлы **VREG.MDF** и **IREG.MDF** из выбранных папок в папку **MODELS** Протеуса. Обращаю ваше внимание, что если вы перед применением **Make Device** запускали симуляцию, изменили напряжение или ток, сменили в окне свойств какие либо параметры, например, **STEP** или **FORMAT**, то модель сохранится именно с этими параметрами, если конечно вы их не исправите на третьей вкладке к другим значениям.

К содержанию

8.4. Модель датчика давления с выходом 4-20мА на основе SETPOINT.DLL.

Я бы изменил своему нынешнему профессиональному эго, если бы не поделился с вами еще одной своей наработкой в области моделирования. По своей сфере теперешней деятельности – проектирование и монтаж автоматики мне постоянно приходится работать с различными датчиками: температуры, давления, расхода и т.д. и т.п. Львиную долю датчиков давления среди всего этого разнообразия занимают датчики с типовым выходным токовым сигналом – 4-20мА. Конечно, можно

использовать в этих целях и обычные примитивы генераторов тока, но это обычно доставляет массу неудобств при моделировании поведения девайса в реальном времени. Каждый раз при смене значения придется полностью останавливать симуляци. «Прикручивание» к ним активных моделей потенциометров тоже не меняет дело к лучшему. В конечном итоге, как правило схемы для моделирования представляла собой нечто вроде изображенных на рисунке 56.

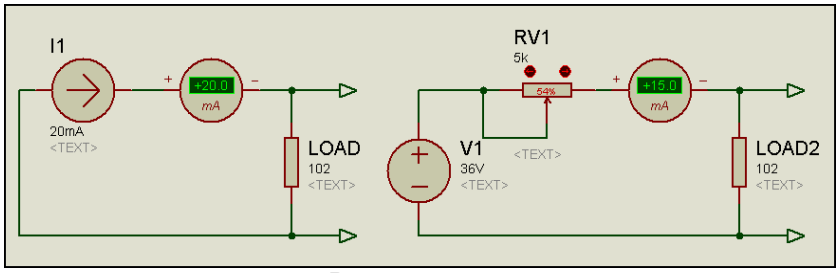


Рис. 56

Мало того, что такие «примочки» в проекте занимают много места, так еще приходится держать перед глазами и таблицу пересчета давления в ток или иметь под рукой калькулятор для этих целей. Естественно меня такое положение дел не устраивало. Ну и сам собой напрашивался прецедент иметь в **ISIS** модель универсального датчика давления для разработки своих устройств. **SETPOINT.DLL** для этих целей подходит как нельзя лучше. Для начала немного арифметики. Предположим, мы имеем датчик избыточного давления от 0 до 100кПа (ну или 0–1кгс/кв.см, что практически тоже самое). Для покрытия этого диапазона нам отводится токовый диапазон от 4мА (соответствует 0) до 20мА (соответствует верхнему пределу), т.е. 20-4=16мА. Разделив 16 на 100 получим, что изменению давления на 1кПа соответствует изменение тока на 0,16мА. Эта ориентировка поможет нам в дальнейшем.
Теперь создаем нашу модель. Художник из меня еще тот, но все же я постарался изобразить нечто напоминающее реальный датчик (Рис. 57).

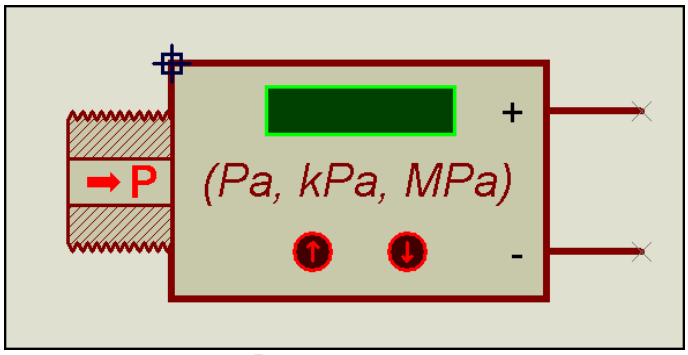


Рис. 57

Поскольку в данном случае графика модели создается «с нуля» остановлюсь на некоторых нюансах. Изображение резьбы в разрезе выполнено с помощью **Closed Path** из левого тулбара, затем в свойствах изображения снимаем галочку с **Fill style** и выбираем режим косой штриховки (Рис. 58). Там же, сняв флажок **Width,** выбираем более тонкий контур для обрисовки объекта - 8th.

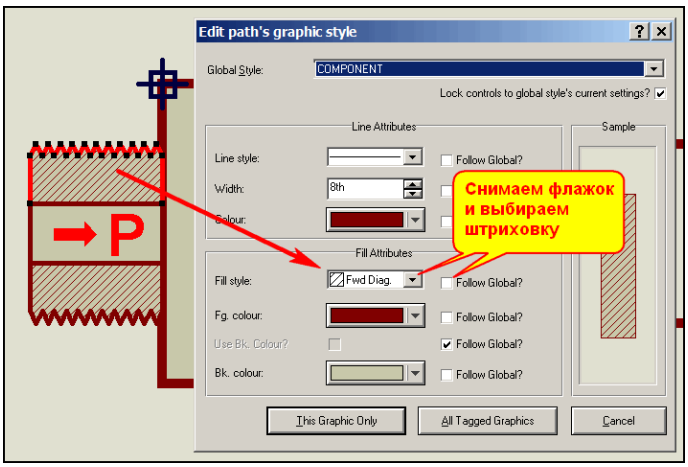


Рис. 58

Индикатор в данном случае тоже нарисован свой. Для этой цели в режиме рисования прямоугольников выбираем в селекторе опцию **INDICATOR** и рисуем наш экран в нужном месте. Чтобы не ошибиться с размерами – лучше поместить рядом что то из реальных моделей, содержащих аналогичный микродисплей, например, – вольтметр. В принципе, вы можете нарисовать индикатор хоть на поллиста проекта, но все же необходимо учитывать, что его параметры взаимодействуют с **SETPOINT.DLL**, а там о них уже до нас позаботился программист, создавший эту библиотеку. Поэтому мы будем иметь тот шрифт и цвет, которые уже есть, ну а максимально индицируемое число, я думаю, вам не составит труда определить самостоятельно, если вспомнить, что мы его в свойствах задаем как **float**.

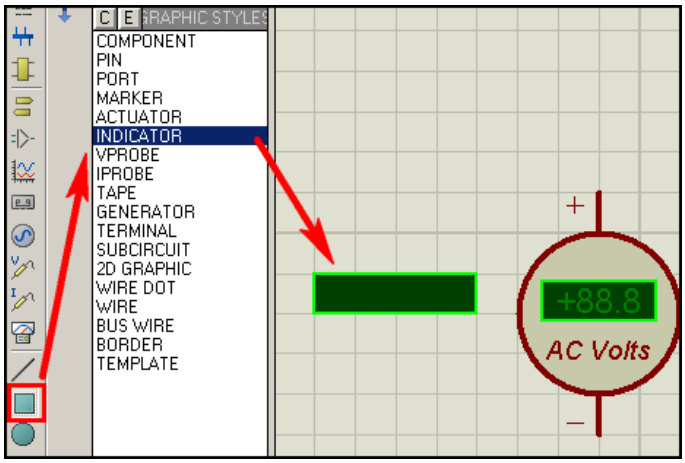


Рис. 59

Ну и, наконец, о маркерах инкремента и декремента. Поскольку это маркеры, мы их и берем из селектора в режиме **2D Graphic Markers Mode** (Рис. 60).

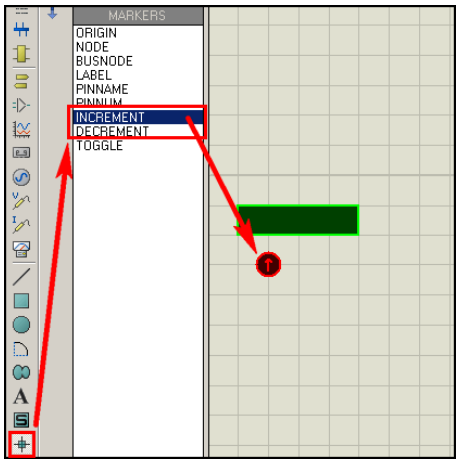


Рис. 60

С остальной графикой, надеюсь, проблем не будет. Надписи помещены чисто в «напоминательных» целях и никаких подводных камней в них не зарыто. Выводам присвоены только имена «**+**» и «**-**» , нумерация нам не нужна, поскольку корпуса эта модель содержать не будет. После того, как прорисована вся графика полностью, выделяем все ее элементы с помощью обводки с зажатой левой кнопки мышки и через **Block Copy** размножаем в двух экземплярах для создания символов. На одной копии удаляем выводы, экран и маркеры инкремента/декремента – это будет база, символ с индексом **_0**, а на другой оставляем только экран и маркер **ORIGIN** – это будет символ с индексом **_1** (Рис 61). Затем из того, что изображено на этом рисунке создаем символы. Я назвал их **SENSOR_0** и **SENSOR_1**. Создав символы, убедитесь, что при переходе в режим символов (кнопка **S** в левом тулбаре) они доступны в селекторе. Только после этого можно приступать к созданию самого девайса из того полного графического изображения вместе с выводами, что я привел на рисунке 57.

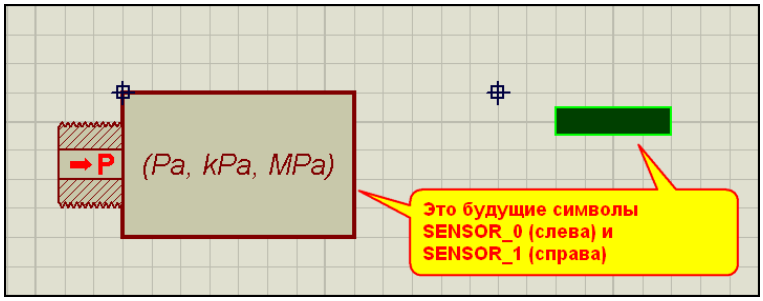


Рис. 61

Сразу же хочу обратить ваше внимание, что вот такой порядок создания графики для активных компонентов заведен и у самих разработчиков в Лабцентре. Т.е. сначала прорисовываем все в совокупности, затем копируем через **Block Copy** и тупо тыкаем левой кнопкой в нужных местах, размножая нужное количество раз полное изображение. Ну, а уж потом, как скульпторы, на копиях отсекаем (двойной клик правой) все ненужное для каждого конкретного символа. При этом минимален риск получить неправильное позиционирование нашего символа относительно маркера **ORIGIN**, если конечно сдуру нечаянно не кликнуть пару раз правой и по нему. Мне этот «фокус» лабцентровцев подсказал Тень, хотя, к тому времени я уже и своими извилинами «дополз», что так получается быстрее. Особенно прирост скорости скажется при создании многоэлементных индикаторов или клавиатур, чем мы займемся чуть позже. Но привыкать к такому порядку я призываю заранее.
Итак, с графикой разобрались. Теперь мой любимый **Make Device**. На первой вкладке задаем имя нашему девайсу и задаем параметры активной графики (Рис. 62). Ну, тут все, как и с источниками, за исключением того, что символы у нас теперь с собственными именами.
Да и на третьей странице мало что изменилось, разве что в **Description** я поставил другие названия, например для самого **SETPOINT** там стоит **Actual Pressure** – текущее давление (Рис. 63).

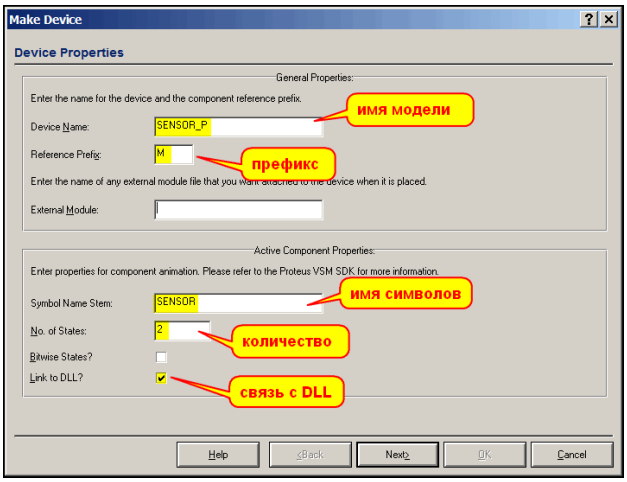


Рис. 62

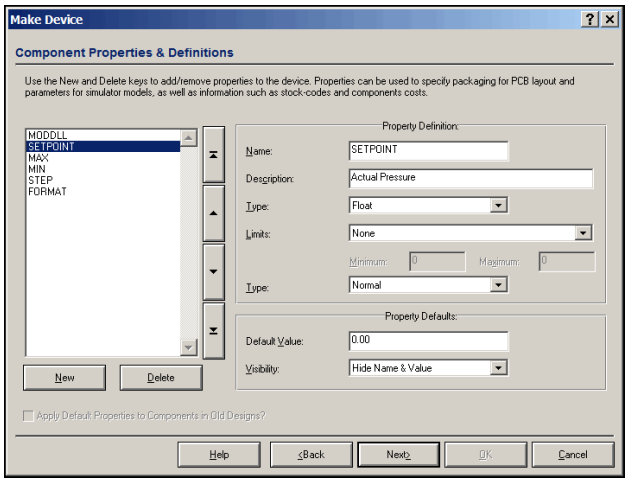


Рис. 63

Я не стану здесь приводить все скриншоты с третьей вкладки, желающие могут открыть пример **Graphic_Child.DSN** в папке вложения **Simple_Model**, войти в свойства или запустить **Make Device** для уже готовой модели и посмотреть на третьей вкладке – что и как я «обозвал». К данному моменту это вы уже должны усвоить это как дважды-два. Можете также перевести **Description** на русский, мне, честно говоря, это делать лень, тем более, что меня никакими коврижками не заманишь пользоваться «русифицированной» версией. Боюсь, что даже если Тень когда-либо сподобится раскрутить свое начальство на выпуск официальной русской локализации интерфейса. После нескольких лет работы настолько привыкаешь к оригиналу, что «уписанное» на место лаконичной английской фразы наше родное, вечно не помещающееся в строку и поэтому урезанное до невозможности типа «Тпр. пр. гр.» расшифровать гораздо труднее, чем полную аглицкую фразу. Ну, это так, небольшое отступление от темы, а мы создаем девайс до конца, сохраняем и как обычно в свойствах «приклеиваем» дочерний лист. На нем создаем внутреннюю структуру нашего девайса, или **Test jig**, как это именуется в оригинальной документации VSM SDK (Рис. 64).

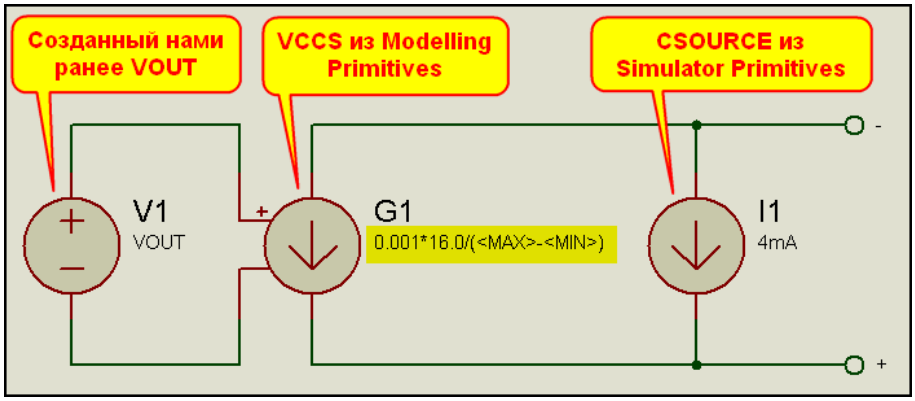


Рис. 64

Давайте разберем эту «джигу» по косточкам. Связка первых двух компонентов аналогична управляемому источнику тока, рассмотренному ранее, вот только у G1 в передаточной характеристике стоит «хитрая формула». Я ее нарочно расписал поподробнее, чтобы было проще объяснять. Первый множитель – **0.001** переводит наше значение тока в миллиамперы. Стоящая в скобках разность **<MAX>-<MIN>** дает нам полный диапазон изменения давления нашего датчика. Причем, мы вольны задавать им любые положительные значения или ноль для минимума. А отношение **16.0** (полный диапазон тока, мы определили его выше) к диапазону изменения давления даст нам коэффициент пересчета давления в ток. Таким образом, **VOUT** будет менять свое значение от **MIN** до **MAX**, а **G1** пересчитывать его в ток от нуля до **16мА**. Источник **I1** с током **4мА** в сумме сдвинет нам диапазон в **4-20мА**, что нам и надо. У кого затруднение с пониманием параллельного включения **G1** и **I1** – попросите у детей физику за 7-й класс и перечитайте законы Кирхгофа. Хотя, это я официально учил в 7-м, а неофициально – самостоятельно в 4-м, а что там сейчас учат, одному директору школы известно. Ох, опять в ностальгию ударился. Идем дальше…
Ну, в общем, идем на **Parent List** запускаем и тестируем нашу модель, используя различные диапазоны значений давления. Все это доступно в упоминаемом выше примере. Я не стал вводить ограничения для **MIN** и **MAX** на третьей вкладке, чтобы вы могли убедиться, что при использовании отрицательных значений для **MIN** мы получим полный бардак на выходе. Если кому то нужна полностью работоспособная модель с такой внутренней структурой, то рекомендую «мэйкануть» ее еще раз и на третьей вкладке для **MIN** выбрать в **Limits** из раскрывающегося списка ограничение **Positive Or Zero** (положительное или ноль), А для **MAX** предел **Positive Non-Zero** (положительное, не ноль). Затем с дочернего листа скомпилировать **MDF** и на третьей вкладке добавить его через **MODFILE**.
Я же, пошел дальше и решил сделать более универсальную модель – датчик лавления/разрежения или на языке КИПовцев – тягонапоромер. Если ей задать нижний предел ноль – это обычный датчик давления, если меньше нуля – тягонапоромер. Поменялся при этом только дочерний лист, а что за «джига» у меня получилась показано на рисунке 65, а также в примере **Uni_Sens_1.DSN** из папки **Uni_Sensor_with_Child**.

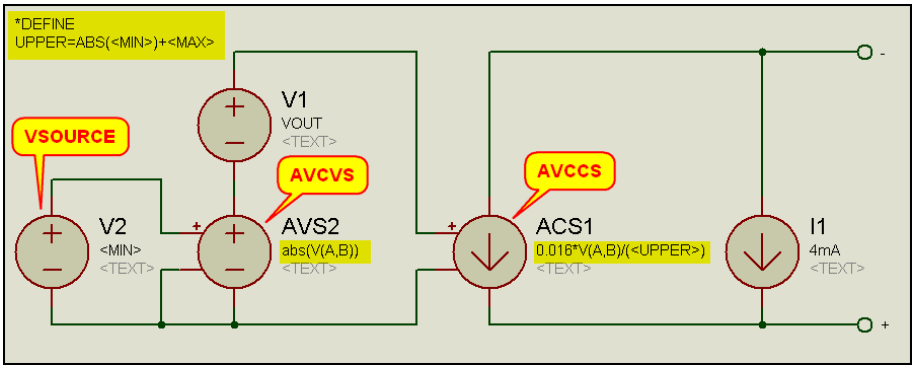


Рис. 65

Какие здесь встретились заморочки. Во-первых потребовалось сдвинуть диапазон VOUT d область положительных значений. С этой целью введены **V2** и **AVS2**. Почему получилось так сложно? Просто для обычных источников функция **abs** (абсолютное значение или по другому модуль) не работает, она предназначена только для **Arbitrary Source** (см. материал из предыдущих частей или HELP по примитивам). С этой же целью и **ACS1** тоже взят **arbitrary**. Там для обычного **VCCS** почему то не работала подстановка диапазона **<UPPER>** из скрипта ***DEFINE**. Формулу я упростил, перемножив заранее **0,001** на **16**. Для этой модели я уже скомпилировал MDF и приложил для

тестирования в папке **Uni_Sensor_MDF**. Там модель уже с присоединенным **MODFILE**. Достаточно протолкнуть для нее **Make Device** ничего не меняя и переложить **MDF** в папку **MODELS** для использования в своих разработках. Вот здесь я для **MAX** ввел **Limits**, а для **MIN** не стал. Вряд ли кому придет в голову делать **MIN** больше **MAX**, хотя у наших доморощенных умов…
На этом абстрактный материал по **SETPOIN**T закончен, но, пожалуй, добавлю далее еще одну модельку из Reality Show, тем более она мне и самому понадобилась.

К содержанию

8.5. **Модель датчика давления** Freescale MPX5010 **из модели** MPX4250.

Предыстория создания этой модели заключается в том, что мне захотелось иметь персональный электронный манометр для контроля давления природного газа на бытовых газовых котлах и промышленных газовых горелках. Есть у нас на работе и поверенные сертифицированные, но бывает так, что под рукой отсутствует, а срочно нужен. Поскольку для этих целей я буду тратить «свои кровные», захотелось создать дешевую и доступную по комплектующим конструкцию. В качестве датчика давления мне приглянулся недорогой **MPX5010** от **Freescale** (он же бывший **Motorola**). Конечно, для портативного прибора лучше было бы использовать более современный **MP3V5050** с трехвольтовым питающим напряжением, но как сказал слесарь Полесов: «при наличии отсутствия пропитанных шпал…» придется обойтись тем, что есть в продаже.
В Протеусе датчики давления представлены всего двумя довольно архаичными моделями в библиотеке **Transdusers=>Pressure**. Это датчик абсолютного давления **MPX4115** с пределом 15…115кПа и датчик относительного давления **MPX4250** с пределами 0…250кПа. Если кто-то первый раз сталкивается с датчиками давления, поясню, что датчики абсолютного давления используются в основном для контроля атмосферного давления в тех же электронных барометрах и метеостанциях (например, любимые нашими синоптиками 760 мм ртутного столба соответствуют 101,308кПа). Датчик относительного давления контролирует превышение давления на его входе относительно атмосферного. Такие датчики применяются для контроля давления в трубопроводах, резервуарах, автошинах и т.п. Мне нужен датчик относительного давления. Предел в 250 кПа меня мало устраивает, мне достаточно 10, но «за основу», как говорили всякие партократы-бюрократы эту модель принять можно, тем более, что сделана она довольно профессионально и заслуживает рассмотрения с точки зрения внутренней структуры. Подключение модели и выдаваемое ею напряжение при минимальном и максимальном давлении показано на рисунке 66.

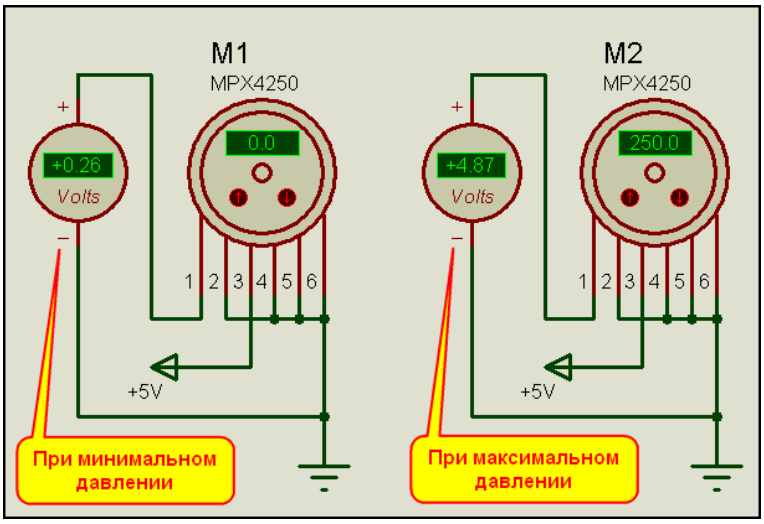


Рис. 66

Для начала заглянем в окно свойств модели этого датчика (Рис. 67) и сопоставим их с даташитом.

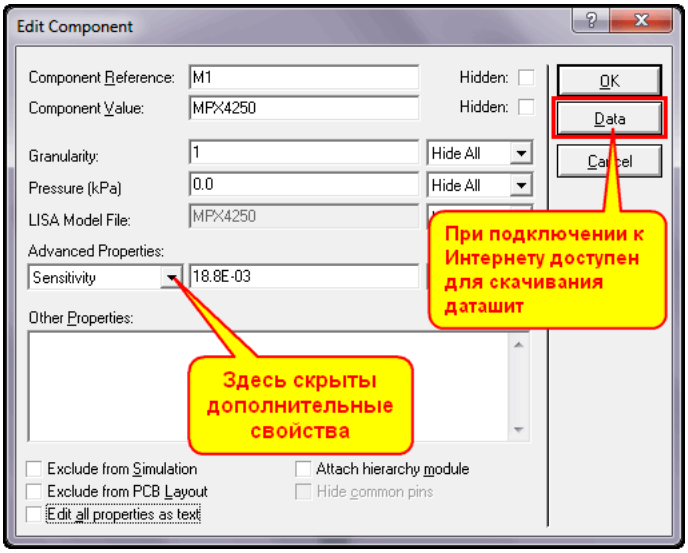


Рис. 67

В основном окне доступны для изменения только два параметра **Granularity** (шаг изменения) и **Pressure** – стартовое значение давления. Модель построена на основе все того же **SETPOINT**, поэтому назначение этих параметров не вызывает затруднений. А вот в раскрывающемся списке **Advanced Properties** «зарыты» именно свойства связанные с даташитом датчика, который, кстати, можно скачать, нажав кнопку **Data**. Рассмотрим их:

Sensitivity – **SENS** (здесь и ниже черным жирным указано как они трактуются в режиме **Edit all Properties as text** и на третьей вкладке **Make Device**). Чувствительность датчика – типовое значение 18,8 мВ/кПа по таблице **OPERATING CHARACTERISTICS** на стр. 2 даташита. Соответственно в Протеусе эта запись будет выглядеть как **18.8E-03**.

Offset – **OFFSET** – смещение. Значение 0,04 фигурирует в формуле **Transfer Function** (передаточная функция) на стр. 3 и 4 даташита.

Pressure Error Band – **PEB** – диапазон ошибки давления максимальное значение 3,45 кПа приведено на графике стр. 4 даташита.

Temperature – **TEMPERATURE** – указанное в свойствах значение TEMP (по умолчанию 27°C) берется из параметров симулятора (меню **System=>Set Simulation Options** вкладка **Temperature**) и позволяет имитировать поведение датчика в зависимости от температуры. О температурном моделировании я уже рассказывал во второй части FAQ.

Теперь восстанавливаем структуру по файлу MDF (Рис. 68).

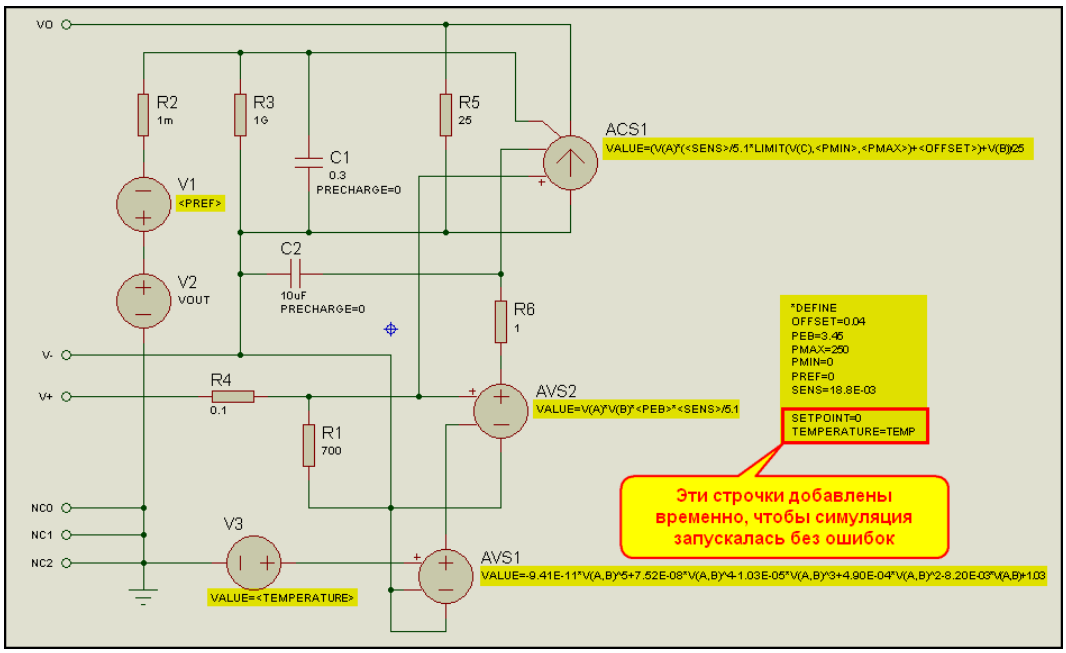


Рис. 68

Так, предчувствую скептические замечания, - опять автор «чудит». Где он взял эту сороконожку – **ACS1** – в примитивах такого нет. Да просто надо внимательно читать HELP по примитивам.

Открываем раздел **The Arbitrary Controlled Source Models - AVS, ACS** и находим там следующий абзац:

> Voltage input values are referred to as **V(A), V(B), V(C)** etc. within the expression, these values referring to the voltages at pins named A, B, C. The form **V(A,B)** is also supported, this meaning the differential voltage between pins A and B.

В вольном переводе это звучит следующим образом:
*Значения входных напряжений описываются в выражении как **V(A), V(B), V(C)** и т.д., где значения соответствуют напряжениям на ножках (пинах) с именами A, B, C. Форма записи **V(A,B)** также поддерживается и соответствует разности напряжений между выводами A и B.*
Как видим, в стандартных примитивах все **Arbitrary Source** с двумя входами – A и B, хотя их может быть … и т.д.. А вот это самое **V(C)** фигурирует в выражении для **AVS1** в MDF модели датчика (выражение выделено желтым цветом на рисунке 68). Но вывода C у примитива не было, пришлось срочно «припаять». Так что ничего сверхъестественного – я просто разобрал (**Decompose**) существующую модель **AVCCS** и приделал к ней еще один вход с именем **C**, ну а потом «мэйк девайснул» свое новое творение. Верхний слева по схеме вывод и есть этот пресловутый C. Для большей компактности схемы мне пришлось отразить модель **AVS1** по вертикали и поэтому выводы входов получились снизу вверх: A, B, C. Именно этот примитив и является основным в модели датчика, и в выражении, описывающем его передаточную характеристику, заложена основная формула преобразования.
Вообще данная модель, с моей точки зрения, обладает даже некоторой избыточностью. Попробую описать назначение остальных узлов схемы, как я их понял. Основным управляющим элементом является все тот же **VOUT V2**, связанный с **SETPOINT.DLL**. Источник **V1** со значением PREF=0 вероятно был заложен для коррекции передаточной характеристики в зависимости от **VOUT**, но поскольку в конечном итоге он оставлен с нулевым напряжением, это явное излишество. Источник **V3**, значением которого является температура, и связанный с ним источник **AVS1** с длинным полиномом в передаточной характеристике, призваны имитировать поведение датчика при температурном анализе. Ну и, наконец, источник **AVS2**, передаточная характеристика которого зависит от напряжения питания, имитирует поведение датчика при изменении питающего напряжения.
Восстановленная из MDF структура **MPX4250** приложена в проекте **Stucture_MPX4250\ Struct_MPX4250.DSN** вложения. В этой же папке приложен и извлеченный из библиотеки **TRXD.LML** файл **MPX4250.MDF**.
Я же приступаю к подгонке структуры под свои нужды, т.е. созданию **MPX5010**. Как я уже предупреждал, источник **V1** отправляю на свалку, соответственно в свойствах упраздняется параметр **PREF**. Максимальное значение в свойствах изменяем с **MAX=250** на **MAX=10**, минимальное остается неизменным. Ошибка давления для новой модели приведена на стр. 6 соответствующего даташита, и составляет: **PEB=0.5** kPa. Несколько сложнее обстоит дело с чувствительностью. В таблице характеристик датчика MPX5010 даташита есть небольшая «очепятка» верхнее значение для **Sensitivity** это как раз 450 mV/kPa, а не какие то там подозрительные mV/mm. Нижнее значение для миллиметров водяного столба указано правильно. Как видите, и на **Freescale** «бывает проруха», это лишний повод мне напомнить, что своя «соображалка» должна быть всегда начеку. Ставим в свойствах **SENS=450E-03.**
Далее необходимо заняться формулами, описывающими передаточные характеристики источников в структуре **MPX4250**. Нам необходимо сопоставить, что там требуется заменить и как в новой модели. **MPX5010**.
Для начала обратим внимание на передаточную характеристику (**Transfer Function**), представленную в даташитах на датчики.
Для **MPX4250** она описывается как: **Vout = Vs x (0.00369 x P + 0.04) ± Error**
где: **Error = Pressure Error x Temp. Factor x 0.00369 x Vs**
Vs= 5.1 ± 0.25 Vdc
Для **MPX5100** она описывается как: **Vout = Vs x (0.09 x P + 0.04) ± Error**
где: **Error = Pressure Error x Temp. Factor x 0.09 x Vs**
Vs= 5.0 ± 0.25 Vdc
Ну, даже беглый взгляд, сразу подсказывает, что там, где в формулах для **MPX4250** стоит напряжение 5,1 для **MPX5010** нам надо поставить ровно 5.
Длинный «член» в AVS1 оставим неизменным, поскольку, если внимательно посмотреть температурные зависимости в даташитах обоих датчиков, то они полностью совпадают. Получившаяся структура представлена в проекте **Struct_MPX5010.DSN** из папки **Stucture_MPX5010** вложения. Далее надо создать графическую модель MPX5010, и поменстить нашу структуру на дочерний лист для тестирования. Мудрить с графикой я не стал. Поступаем просто: берем модель **MPX4250** и применяем к ней **Make Device**. На первой вкладке переименовываем в **MPX5010**, на

третьей удаляем, ставший ненужным параметр **PREF**, а также временно удаляем **MODFILE**, позже мы присоединим новый. Там же правим значения остальных свойств: **MAX**, **PEB**, **SENS**, как было указано выше.Ну и перед сохранением на последней вкладке в окне **Device Description** исправляем предел с 250 на 10 kPa, чтобы потом не путаться самим. Теперь как обычно помещаем нашу модель в проект, присоединяем к ней дочерний лист и помещаем туда нашу структуру. Этот вариант представлен в папке **MPX5010_Child** вложения.

Тестируем, затем с дочернего листа компилируем MDF и еще раз пройдясь через **Make Device** присоединяем на третьей вкладке наш **MODFILE**. Готовый вариант с MDF представлен в папке **MPX5010_MDF** вложения.

Ну и под занавес я не удержался от того, чтобы проверить – как влияет вот тот длинный «член»-полином температурной зависимости в **AVS1**. Этот проект представлен в папке **MPX5010_TEMP**. Для того, чтобы получить зависимость выходного напряжения от температуры, я воспользовался графиками **DCSWEEP**. В свойствах датчика для **Temperature** подставляем вместо **TEMP** значение **X**, а в свойствах графика для X задаем значения от -40 до 125 – как в даташите. Я сделал два графика при нулевом и максимальном давлении. Они представлены на Рис. 69.

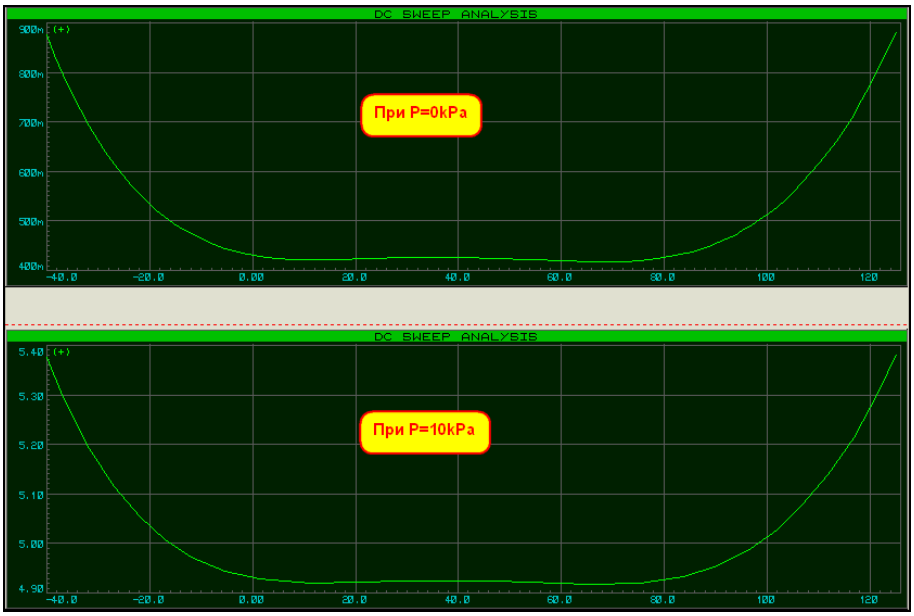


Рис. 69

Ну, по большому счету, можно сказать, что «многочлен» работает – налицо характерные завалы вверх ниже нуля и выше 80 градусов, которые есть и на графике в даташите. Даташиты на датчики **MPX4250** и **MPX5010** также присутствуют во вложении, так что можете убедиться самостоятельно. Я же на этом заканчиваю материал с моделированием на базе **SETPOINT** и перехожу к другим программным моделям на основе DLL.

К содержанию

8.6. LEDMPX.DLL – **основа всех активных «светящихся» цифровых индикаторов в** ISIS.

«Все-таки модель индикатора в протеусе штука глючная.»

shamber (30.01.2011)

Наконец-то я добрался до подробной разборки с этой библиотекой, чтобы раз и навсегда положить конец утверждениям «чайников» о глючности сегментных индикаторов в ISIS. Видимо информации в первой части FAQ по динамической индикации оказалось недостаточно. Будем «жевать» дальше.

Итак, из самого названия библиотеки можно вынести уже достаточно информации. **LED** говорит нам о том, что предназначена она для создания всевозможных светодиодных индикаторов, а **MPX** – принятое «за бугром» сокращение слова **multiplexing**. Вот именно на этот последний термин почему то многие откровенно «наплевали и забыли», а зря. Приведу дословно одно из толкований, встречающихся в Интернете: *«Процесс объединения отдельных потоков или каналов в один логический поток данных таким образом, что они позднее могут быть восстановлены в прежнем виде без ошибок. Двумя наиболее широко используемыми технологиями мультиплексирования являются частотное разделение каналов, когда для передачи различных сигналов используются несущие с различной частотой, и временное разделение каналов, когда отдельные сигналы передаются в свои временные промежутки».* Пожалуй, такая трактовка наиболее полно отражает то, что уже в самой библиотеке заложено мультиплексирование, т.е. разделенная во времени обработка сигналов. И об этом забывать никак нельзя, особенно когда вы

на эту динамику накладываете еще и свою, пытаясь симулировать динамическую индикацию со СВОИМИ временными параметрами, подчас, далеко не идеальными. Вот и попробуйте мысленно представить себе – что должно получиться в итоге, когда вы наложите друг на друга:

- Ваши собственные импульсные последовательности сканирования катодов индикаторов (раз) и анодов индикаторов (два);
- Времянки самой **LEDMPX** – по умолчанию **Minimum Trigger Time** установлено, как 1мсек (три)
- Параметры мультипликации самого симулятора на экране – по умолчанию в **System => Set Animation Options** частота смены картинок **Frames per Second** равна 20 (четыре).

Для того чтобы эти четыре составляющие дали адекватный результат нужно очень внимательно соотнести их параметры и универсального рецепта тут нет и быть не может. Всегда требуется индивидуальный подход, поскольку один применяет динамическую индикацию с частотой 25 Герц, другой – 40, а третий может вообще задрать до 200. Поэтому, только при грамотной установке временных параметров симулятора и самой модели на основе **LEDMPX.DLL** в соответствие с выходными параметрами симулируемой динамической индикации вы увидите именно ту картинку, которая должна быть в реальности. Ну и конечно не стоит забывать о гашении при переключении разрядов индикатора, о котором шла речь в первой части FAQ. На этом я заканчиваю «лирическое отступление» и перехожу непосредственно к самой **LEDMPX.DLL** и моделям на ее основе.

Определить, что модель индикатора основана на **LEDMPX.DLL** совсем не сложно, для этого достаточно обратить внимание на правый верхний угол окна **Pick Devices** (Рис. 70). Кроме того, у многих индикаторов в названии встречается само сочетание MPX, например, **7SEG-MPX4-CA**.

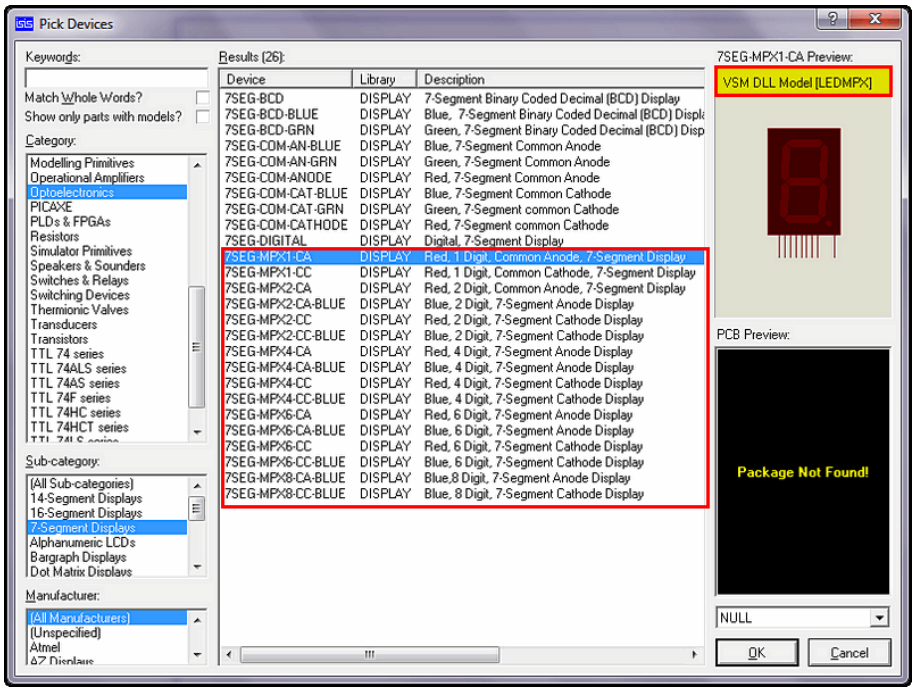


Рис. 70

Все модели на основе этой DLL расположены в категории **Optoelectronics**, а более конкретно в четырех ее подкатегориях: **14-Segment Displays**, **16-Segment Displays**, **7-Segment Displays**, **Dot Matrix Displays**. На рисунке 71 приведены наиболее характерные представители этих категорий, причем во включенном режиме. Я бы хотел здесь еще раз напомнить, чтобы вы не путали одиночные семисегментные индикаторы на основе **LEDMPX** со Schematic, которые мы рассматривали ранее – ведут эти модели себя совершенно по-разному. Характерная особенность одиночных индикаторов на основе **LEDMPX**, который их сразу выдает визуально, – уменьшенный до 50th шаг выводов. Эти модели и 14-ти 16-ти сегментные индикаторы появились только в последних версиях Протеуса, если не изменяет память, то с версии 7.4. Тогда же кардинально была изменена и сама **LEDMPX.DLL**, хотя **Help** Протеуса об этом скромно умалчивает вплоть до версии 7.8, видимо Лабцентр придерживает эту «фишку» на будущее.

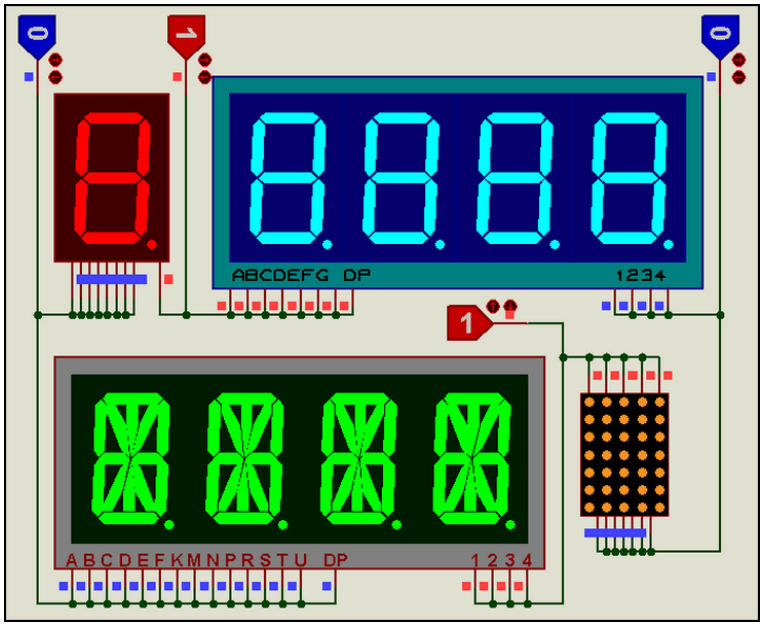


Рис. 71

До хелпа по мультиплексированным LED дисплеям можно добраться как через:
ПУСК => Все программы => Proteus 7 …. => Proteus VSM Model Help => LED and LCD Display Models,
так и непосредственно из окна свойств любой модели вышеупомянутых индикаторов, нажав кнопку Help справа. Раздел, посвященный этим моделям, называется **MULTIPLEXED LED DISPLAY**. Я не стану здесь дословно переводить весь этот раздел, но характерные особенности в моей трактовке изложу ниже.
LEDMPX.DLL может использоваться для моделирования произвольной светодиодной матрицы состоящей из знакомест (цифр или строк матричного индикатора) и сегментов (столбцов матрицы). Характерными приложениями для данной библиотеки является моделирование многоразрядных сегментных индикаторов и точечных светодиодных матриц. При этом обеспечивается двумерный массив строки (или знакоместа сегментного индикатора) обозначаются цифрами – **1, 2, 3** и т.д. а столбцы (или отдельные сегменты цифры) индикатора обозначаются буквами латинского алфавита – **A, B, C, D** и т.д. В оригинальном хелпе указано, что максимальное количество строк и столбцов может быть до 8, т.е. максимум можно было смоделировать 64 светящихся точки (сегмента). Так и было в действительности до тех пор, пока **LEDMPX.DLL** не была расширена под 14-ти и 16-ти сегментные индикаторы. Библиотеку расширили, а хелп оставили нетронутым. Но взгляните на 14-ти сегментный индикатор на рисунке 71 – он явно противоречит хелповской догме. Буковок, то бишь, столбцов матрицы (сегментов) – 14, а никак не 8, да еще и точка **DP**. Подозреваю, что на данный момент матрица имеет ограничения как минимум 16x16 строк/столбцов, проверять на максимум пока лень, дождемся пока Лабцентр «разродится» свежим хелпом.
Второй важной особенностью моделей на основе **LEDMPX.DLL** является то, что модели обладают чисто цифровыми свойствами. И об этом разработчики честно сообщили в help:

> The LEDs are modelled at the digital level only; there is no attempt to simulate their analogue behaviour (i.e. diode characteristics).

Но, первая из бед России в данном случае превалирует над дорогами, мы и по-русски то с ошибками, где уж дорасти до заглядывания в аглицкие хелпы. Вот и плодятся на форуме утверждения, подобные «эпиграфу» данного параграфа.
Ну, что-ж, давайте еще раз разжевывать эту жвачку. Поскольку модели на основе **LEDMPX.DLL** не обладают аналоговыми свойствами, они по определению цифровых моделей обладают бесконечно большим сопротивлением и не потребляют ток от источников сигнала. Проверяем данный постулат. Взгляните на рисунок 72. Тут как в избитой телерекламе, «трюк самостоятельно не повторять, опасно для жизни». Какой реальный индикатор такое издевательство выдержит? А виртуальный работает, и даже дым от экрана дисплея не идет. Обратите также еще раз внимание на нулевые показания амперметра, сейчас мы и это «разжуем и проглотим».

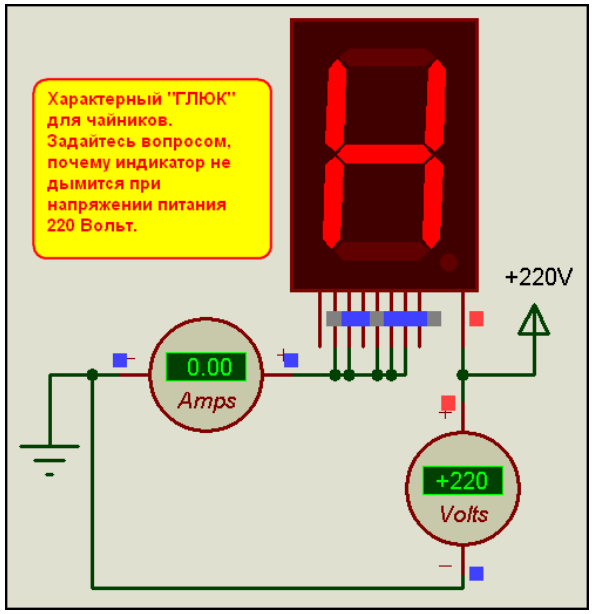


Рис. 72

Опять-таки характерной ошибкой пользователей «с дудочкой» является попытка управлять **ЦИФРОВОЙ** моделью индикатора с помощью аналоговых элементов, например, транзисторных ключей. Ну, в схеме, которую взяли из книжки, журнала, Интернета так нарисовано, значит должно работать. В жизни да, а вот в симуляторе…. Обратимся к следующему примеру (Рис. 73).

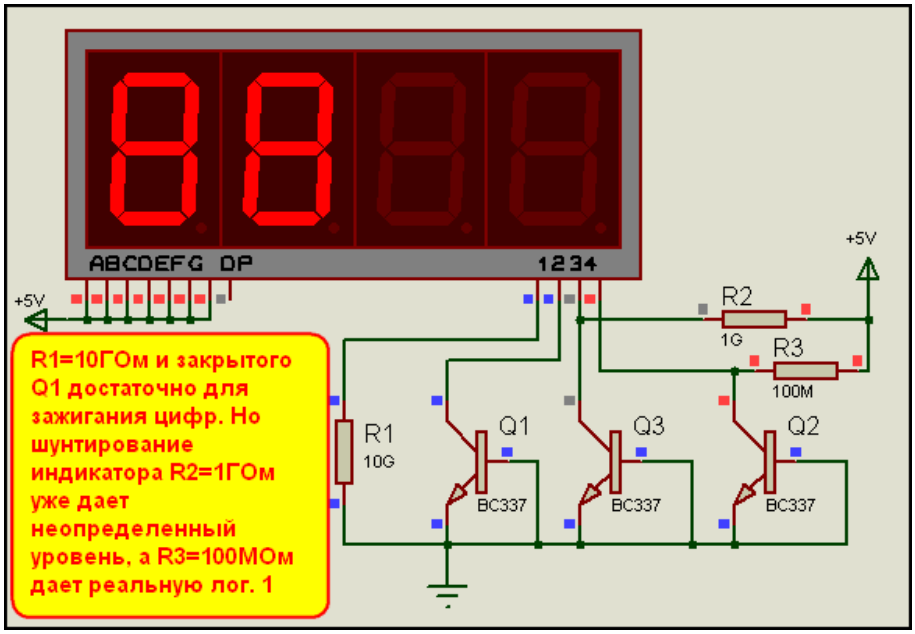


Рис. 73

Здесь у индикатора с общим катодом все четыре цифры включены по-разному. Мы видим, что даже резистора **R1** сопротивлением 10гигаОм в цепи катода достаточно, чтобы спровоцировать зажигание сегментов цифрового индикатора (левая цифра), что уж говорить о закрытом транзисторе **Q1** с заземленной базой (вторая слева цифра). Но, стоит зашунтировать цифру сопротивлением **R2**, пусть и очень большим в 1гигаОм, но аналоговым элементом (третья цифра) и на ее ножке и коллекторе **Q3** уже появился неопределенный уровень, а сопротивление в 100мегаОм (четвертая цифра) уже дает вполне реальную логическую единицу на коллекторе **Q2**. Именно этих цифровых парадоксов модели и не учитывают начинающие пользователи. Причем, в отговорки иногда пускается и такое – у меня нет аналоговых элементов, я питаю через ключ, например, столь у нас популярный **ULN2003**. Да модель то этого ключа в Протеусе схематичная и с аналоговыми свойствами. По-другому и быть не может, вы загляните в даташит этого ключа (Рис. 74) – все те же транзисторы на выходе. Поэтому подходы к симуляции здесь могут быть разными. Если вы собираетесь трассировать печатную плату в **ARES**, то надо делать два проекта – один для создания печатной платы с полным набором схемотехники, а второй для отладки – в нем «все лишнее за борт». Если же Вам необходимо только проверить проект в симуляторе, а печатную

плату вы будете делать в чем то более продвинутом, чем **ARES**, то можно сразу пойти по упрощенному пути. В частности те же транзисторные ключи на рисунке выше можно заменить обычными цифровыми инверторами, тогда и шунтирующие резисторы не нужны, и работать все будет как надо. Ну а если все-таки надо имитировать многоразрядный индикатор и аналоговые ключи, то придется идти на такие вот навороты, как на рисунке 73. Хотя, уж если подходить к этому вопросу корректно, то надо параллельно сегментам вешать не резистор, а диод и резистор, чтобы обеспечить проводимость в одном направлении – это будет ближе к реальности.

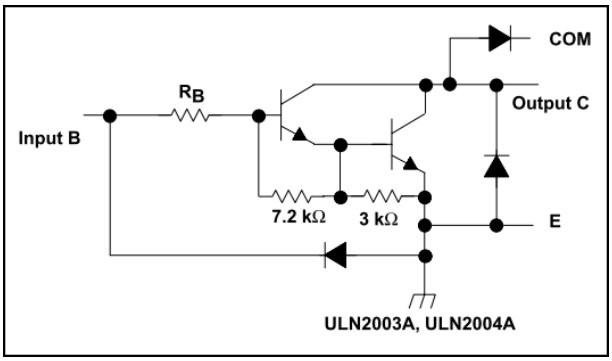


Рис. 74

Итак, надеюсь, больше на том, что все рассматриваемые далее индикаторы являются исключительно цифровыми моделями, останавливаться не надо.
Я же хочу заострить ваше внимание на последнем параграфе хелпа по **LEDMPX**, посвященном временным характеристикам и полярности сигналов, подаваемых на выводы индикатора для зажигания того или иного сегмента. Здесь тоже есть небольшая неточность в хелпе, кочующая из версии в версию, вплоть до 7.8. Специально цитирую именно оттуда:

> By default, a segment is drawn as lit if the corresponding row and column pins are both high for more than 1us during a given animation frame (typically 50ms). This gives a crude but effective representation of persistence of vision; in the real world, the segments are driven with a heavy current for short period of time, and the eye averages out the brightness.

Ну и почти дословный, слегка «облагороженный», с точки зрения фразеологии мой перевод:
По умолчанию, сегмент принимается, как светящийся, если соответствующие выводы строки и столбца находятся в состоянии логической единицы более чем 1 микросекунда в течение данного фрейма мультипликации (типовое значение 50мсек). Это позволяет грубо, но эффективно представить элемент, как светящийся, в реальности же сегменты управляются короткими сильноточными (от слова ток) импульсами, а глаз воспринимает среднюю яркость свечения.
Это как раз то, о чем я толковал в начале данного параграфа, за исключением одного маленького казуса. Если вы заглянете в свойства любого индикатора, то по умолчанию **Minimum Trigger Time** там равно **1ms**, а не **1us**, как написано в help. Это существенная разница, и зачастую простое уменьшение значения к рекомендуемому в help может значительно поправить вид индикации.
Теперь о полярности сигналов. Я вовсе не ошибся в переводе, и в самом Help так же сказано. Сегмент должен светиться при высоких уровнях на выводах соответствующих строки и столбца. Это если в свойствах модели не проинвертированы значения сигналов на одной из групп – столбцы или строки. А они обязательно проинвертированы, именно этим и достигается имитация индикаторов с общим катодом или с общим анодом и больше ничем. Заглянем для примера в окно свойств любого многоразрядного индикатора с общим анодом и поставим там галочку **Edit all properties as text** (Рис. 75).

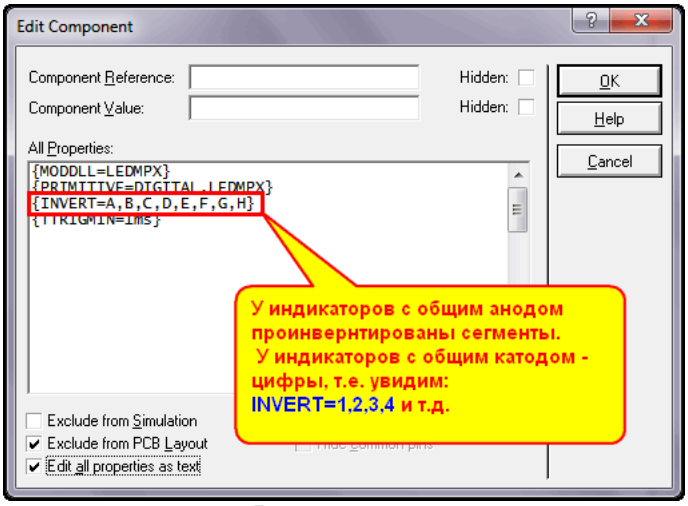


Рис. 75

Ну вот, именно об этом и идет речь в последнем абзаце Help. Мы же для себя делаем очень важную «зарубку» в памяти – для индикаторов на основе **LEDMPX.DLL** должны быть проинвертированы в свойствах состояния тех выводов (пинов), со стороны которых предусматривается управление нулевым логическим уровнем. Т.е. для индикаторов с общим анодом должны быть проинвертированы состояния сегментов:

INVERT=A,B,C,D,E,F,G,H

а для индикаторов с общим катодом состояния цифр (знакомест):

INVERT=1,2,3,4 и далее по количеству цифр.

Ну и заключительные комментарии из Help для **LEDMPX.DLL**, касающиеся непосредственно активной графики, к созданию которой мы перейдем в следующем параграфе. Индикаторы на основе данной библиотеки являются бит-зависимыми (с этим термином мы уже встречались при создании семисегментного **Schematic** индикатора). Общий символ (опять оттуда же, символ с индексом **_C**) определяет ширину знакоместа. Более понятно это станет при создании собственного индикатора, чем мы далее и займемся.

К содержанию

8.7. Активная графика сегментных индикаторов на основе LEDMPX.DLL. Трехразрядный индикатор из четырехразрядного.

Чтобы лучше понять, как строится графика индикаторов на основе **LEDMPX.DLL**, займемся непосредственно практикой. В качестве подопытного «символа года» воспользуемся красным четырехразрядным индикатором **7SEG-MPX4-CA**. Добываем его из библиотеки, помещаем в поле проекта и производим над ним варварское действие с помощью молотка, т.е. **Decompose** этот девайс. Теперь, если переключить левый тулбар в режим **Simbol Mode** (кнопка S), то в селекторе мы увидим полный набор символов из которых состоит наш индикатор (Рис. 76).

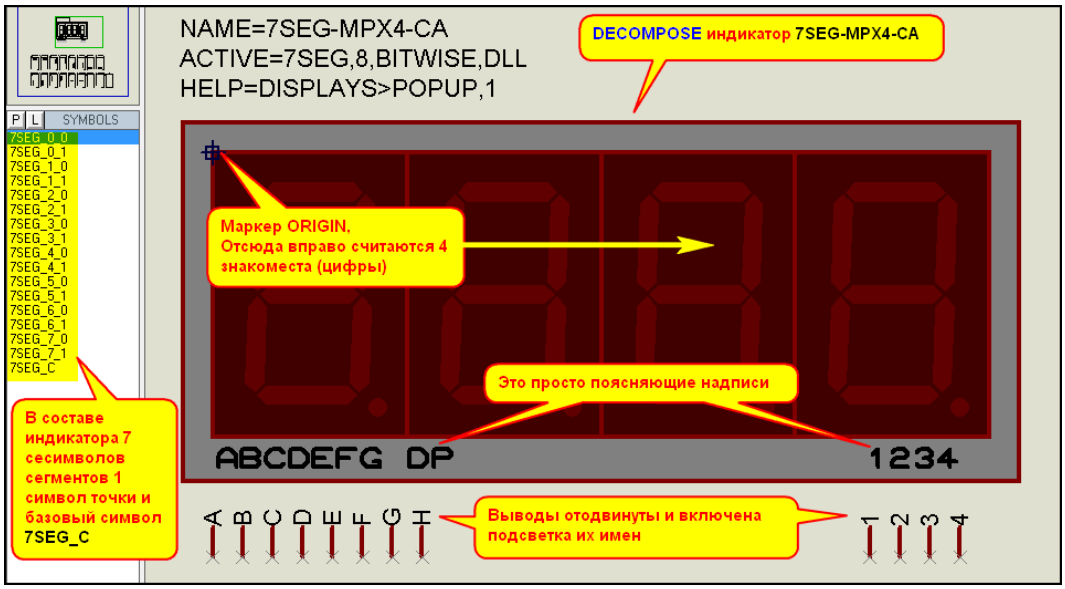


Рис. 76

Ничего нового и интересного вы тут не найдете, все это мы уже проходили в **п.7.2** нашего FAQ. Те же символы **7SEG**, погашенные с индексом **_0** на конце, зажженые – с символом **_1**. Всего в составе индикатора 7 символов для сегментов, начиная с нулевого, и символ десятичной точки – **7SEG_7**. Имеется и базовый символ (подложка) с индексом **_C**. На рисунке 77 я разобрал все эти символы с помощью Decompose, а также нанес для каждого синим пунктиром границы подложки, чтобы вы имели представление – как они построены. И тут для нас пока ничего нового нет.

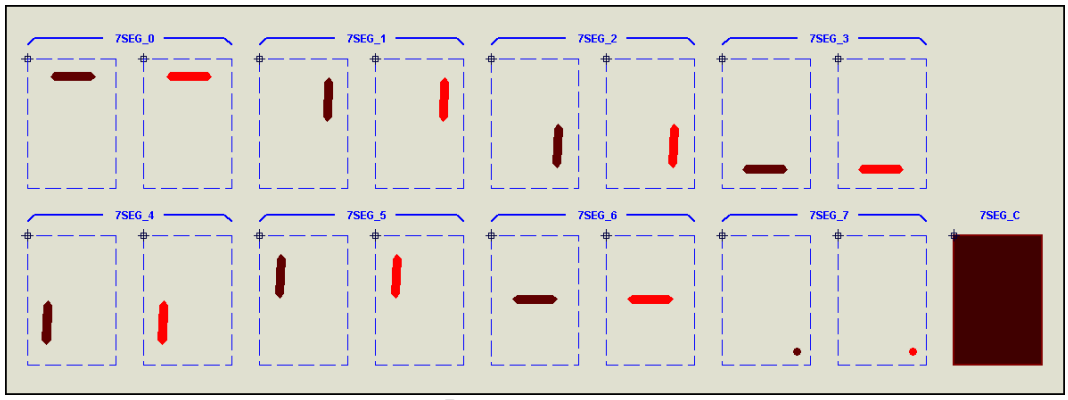


Рис. 77

Но вернемся все же к предыдущему рисунку 76 и обратим внимание на положение маркера ORIGIN – в левом верхнем углу самого левого знакоместа. Этот момент очень важен! Отсюда начинается отсчет знакомест в правую сторону. Ширина каждого знакоместа равна ширине символа подложки (**Common**) с индексом **_C**. Сколько таких «подложек» мы расставим вправо, столько и будет знакомест (столбцов) в нашем индикаторе и столько же должно быть выводов для их сканирования – выводов с именами **1, 2, 3** и т.д. Я на этом рисунке специально отодвинул выводы от тела компонента и включил подсветку их имен. Обратите также внимание, что черные надписи, обозначающие имена выводов остались при этом на месте, потому что это просто текстовые надписи и ничего более, а в модели их применили чисто в справочных целях, чтобы пользователь не заблудился – где какой вывод. Именно поэтому они нечетко совпадают по позициям с соответствующими выводами, а выводу **H** соответствует в надписи обозначение **DP** (от английского Decimal Point – десятичная точка). Это первый этап нашего эксперимента и я приложил его в папке **Step1** вложения.

Ну а теперь перейдем ко второму этапу и займемся чудесным превращением индикатора в трехразрядный. Делается это очень просто и не отнимет у нас много времени. Выполняем поэтапно с иллюстрациями. Для начала удаляем графическое изображение подложки самого правого, т.е. четвертого знакоместа (Рис. 78). Обратите внимание, что я не зря употребил термин «графическое изображение». Внутри графики компонента используются не символы, которые сами по себе содержат внутри маркер **ORIGIN**, а именно простые графические изображения – в данном случае закрашенный прямоугольник определенной ширины. Это я к тому, что если вы собрались наоборот добавлять знакоместа (столбцы), то надо либо скопировать соседнее вместе с графическими изображениями сегментов, либо вытащить на свободное место символ подложки и предварительно его разбить (**Decompose**), чтобы убрать оттуда маркер **ORIGIN**.

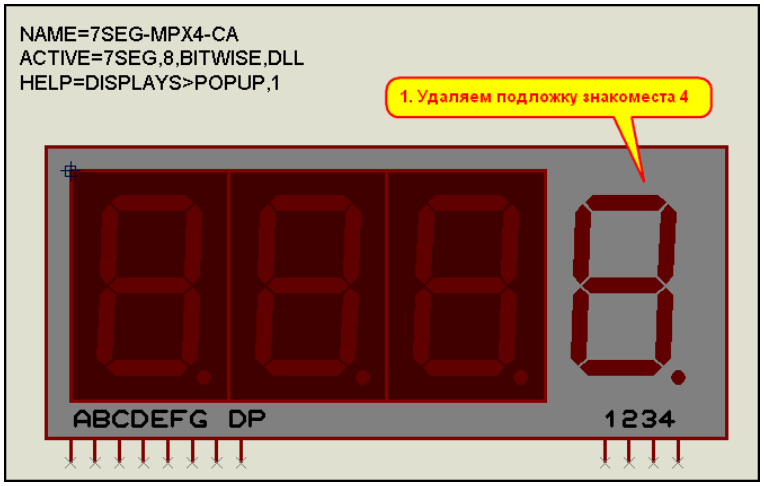


Рис. 78

Теперь выделяем и удаляем графические изображения сегментов и точки того же знакоместа (Рис. 79). Сейчас это сделать проще с помощью «лассо», т.е. обводим их с зажатой левой кнопкой мышки и потом жмем клавишу **Delete**. Теперь уже нет риска зацепить выделением что-нибудь нужное, они достаточно отдалены от соседнего знакоместа.

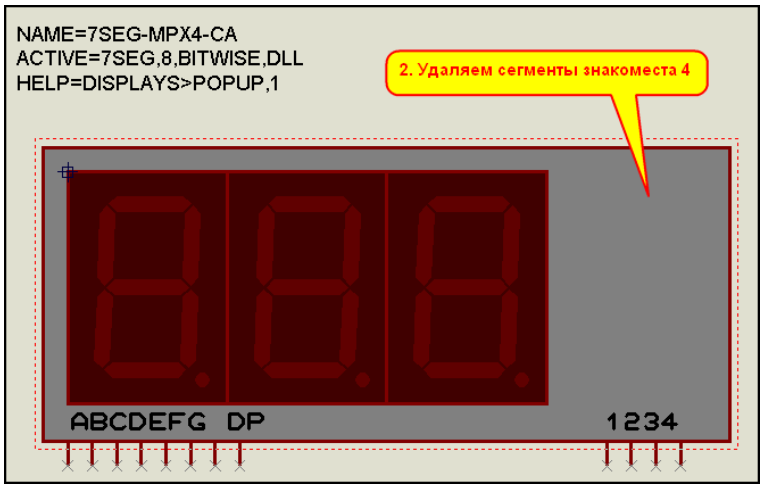


Рис. 79

Далее нам предстоит сдвинуть нужные нам выводы с именами 1, 2, и 3 левее, чтобы они оказались под третьей цифрой. Туда же сдвигаем и надпись «**1 2 3 4**» и редактируем ее, убрав ненужную цифру 4. Эта операция представлена на рисунке 80.

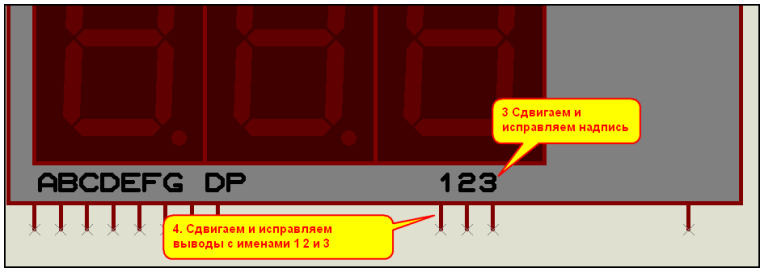


Рис. 80

Теперь нам осталось сдвинуть влево правую границу серого прямоугольника, обрамляющего весь наш индикатор и удалить оставшийся «за бортом» вывод с именем 4 (Рис.81).

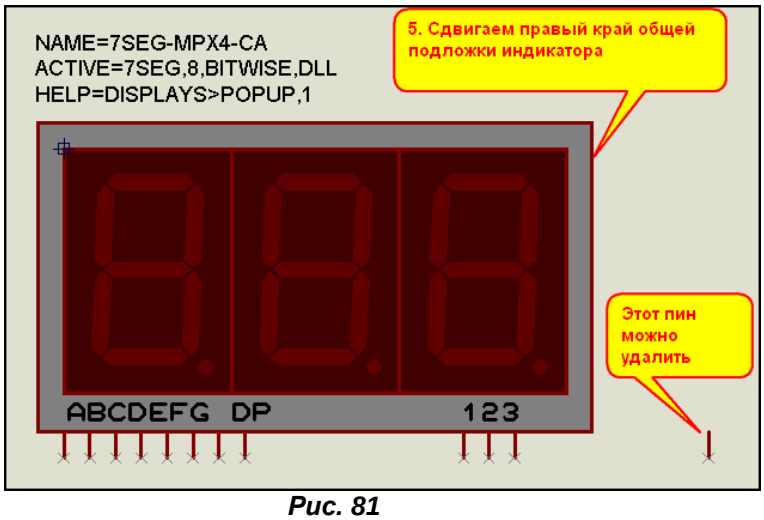


Рис. 81

Все, коррекция графики на этом закончена. Выделяем с помощью лассо всю нашу модель, включая скрипт, и нажимаем любимую кнопку **Make Device** (Рис. 82).

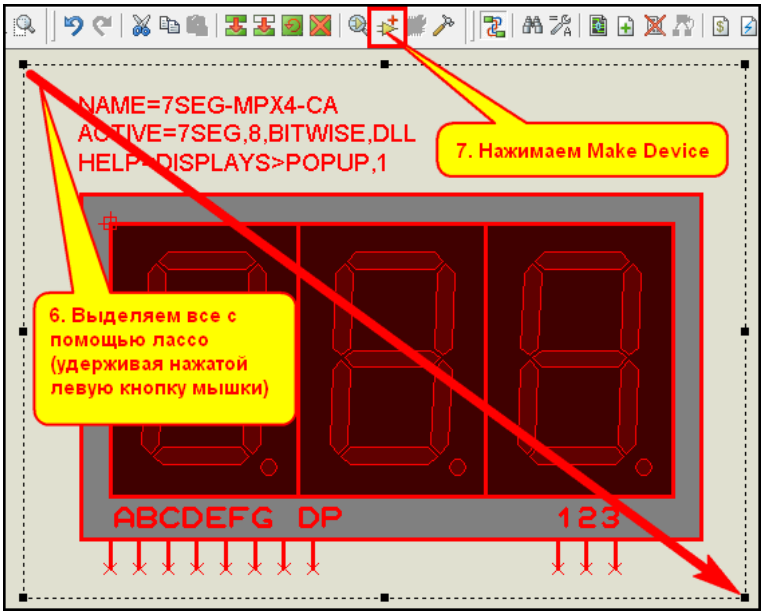


Рис. 82

Если мы не забыли зацепить скрипт, то на первой вкладке процедуры **Make Device** нам достаточно только поправить имя нашего девайса, т.е. изменить 4-ку на 3-ку (Рис. 83). Поскольку мы не меняли количество и имена символов, и они присутствуют в селекторе, то в поле **Active Component Properties** все остается без изменений. Достаточно только убедиться, что включены флажки **Bitwise States** и **Link to DLL**.

Немного остановлюсь на третьей вкладке. Здесь тоже все остается без изменений, но на будущее, если вам надо создать такой же индикатор, но с общим катодом, то достаточно еще раз запустить процедуру **Make Device** для того индикатора, который мы сейчас сделаем. На первой вкладке изменить в имени **CA** на **CC** (от английского Common Catode), а здесь для свойства **INVERT** указать не сегменты, а знакоместа, т.е. в поле **Default Value** для трехразрядного индикатора вписать **1,2,3** (Рис. 84).

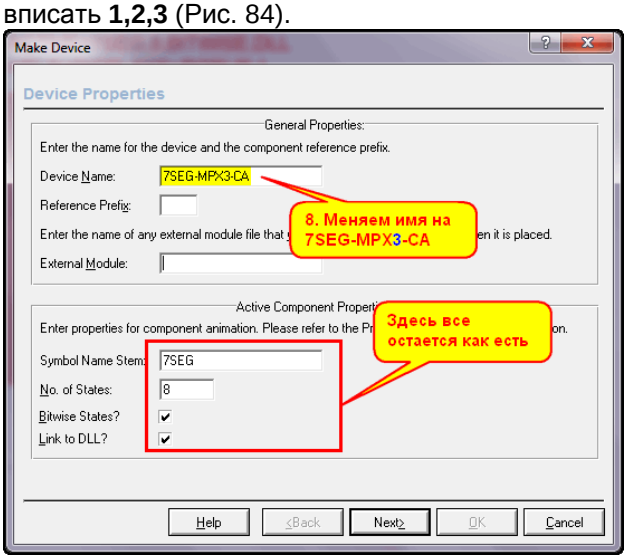


Рис. 83

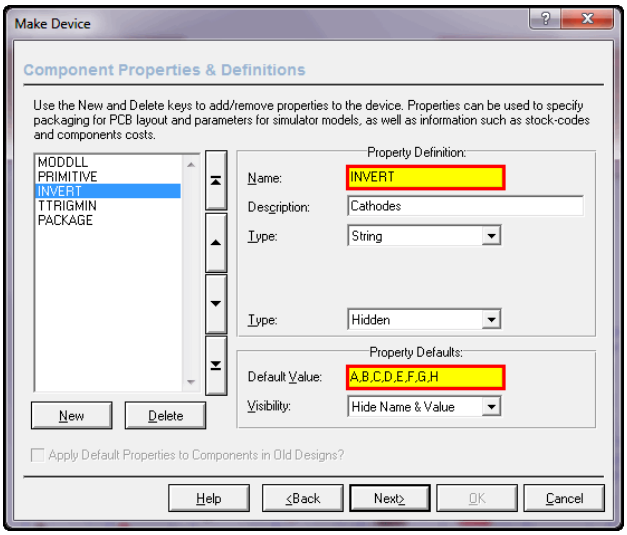


Рис. 84

Ну и теперь доходим до последней вкладки, приводим в **Description** в соответствие количесво цифр нашего индикатора и сохраняем (Рис.85). У меня для сохранения сейчас открыта только **USRDVC**, но можете заранее открыть для записи в менеджере библиотек ту, в которой хранятся семисегментники и сразу сохранить наш девайс там. Библиотека называется **Display** и на сегодняшний день там свободно 13 позиций.

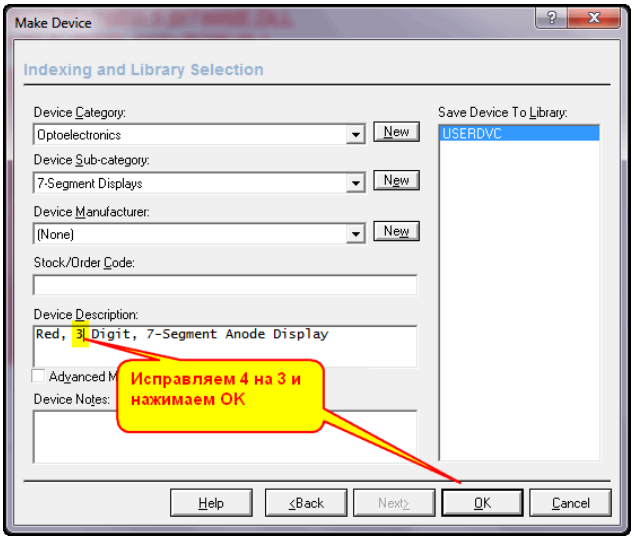


Рис. 85

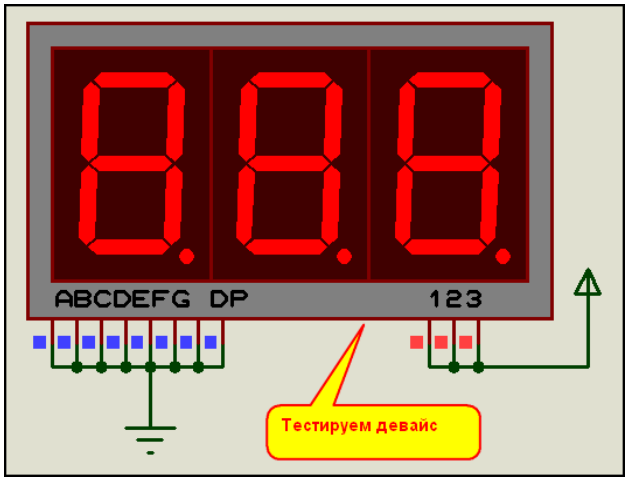


Рис. 86

Нам осталось запустить тестирование нашего устройства и убедиться, что оно работает (Рис. 86). Второй этап нашего эксперимента на этом завершен. Он находится в папке **Step2** вложения. Для того, чтобы сохранить трехразрядые индикаторы в своих библиотеках достаточно пройти процедуру Make Device ничего не меняя для уже готовых индикаторов с общим анодом и общим катодом из этого проекта. Как видите, здесь не требуется даже использовать дополнительные файлы моделей, вся необходимая информация уже содержится в свойствах самой графической модели. Аналогичным образом можете создать самостоятельно индикаторы с другим цветом свечения и с другим количеством разрядов. Необходимо только помнить, что для **LEDMPX** знакоместа располагаются в один ряд по горизонтали слева направо от маркера **ORIGIN** и постараться не превысить общее допустимое количество строк (сегментов) и столбцов (знакомест) матрицы **LEDMPX.DLL**. Из личного опыта сразу могу предупредить, что если допустимое количество будет превышено, то в вашем девайсе будут светиться одновременно не один сегмент (точка), а сразу два. Далее мы поупражняемся с точечными матрицами на основе LEDMPX и на этом закончим материал по этой DLL.

К содержанию

8.8. Активная графика точечных матриц на основе LEDMPX.DLL. Идем на рекорд – матрица 16x16.

В исходном варианте библиотеки **Optoelectronics** в ISIS точечные светодиодные матрицы (**Dot Matrix Displays**) представлены двумя наиболее распространенными видами: 5x7 и 8x8 точек. Это матрицы синего, зеленого, красного и оранжевого цвета. Все эти матрицы построены на основе **LEDMPX.DLL.**

Интерес разработчиков к использованию точечных светодиодных матриц в последнее время значительно вырос. Они широко используются как в рекламных проектах, так и для создания всевозможных индикаторов типа бегущей строки. Наряду с уже давно заполонившими наш рынок матрицами фирмы **Kingbright**, появились в продаже матрицы китайских **Betlux**, **Ningbo Foryard Optoelectronics**. Выпускает матрицы и компания **Rohm**. Причем в продукции этих фирм можно встретить матрицы форматов 5x8, 5x9, 6x8, 16x16 элементов, которые отсутствуют в библиотеках ISIS. Сразу сделаю оговорку для нетерпеливых – построить двухцветную, а уж тем более RGB-матрицу на основе **LEDMPX.DLL** не выйдет. Вспомните сам принцип работы **LEDMPX** – точка (сегмент) может иметь только два состояния (**_0** или **_1** в конце обозначения соответствующего символа) в данный момент времени, определяемый знакоместом (цифрой). Но, тем не менее, облегчить себе жизнь при разработке проектов с использованием одноцветных матриц мы попробуем.

Итак, в качестве примера создадим зеленую матрицу формата 16x16. Заодно и проверим, насколько расширились возможности **LEDMPX.DLL**. Берем зеленую матрицу 8x8 – модель **MATRIX-8X8-GREEN** и разбираем на запчасти с помощью **Decompose** (Рис. 87). Если перейти в режим отображения символов, то в селекторе мы обнаружим восемь символов точек с индексами с нулевого по седьмой и общий базовый символ с индексом **_C**.

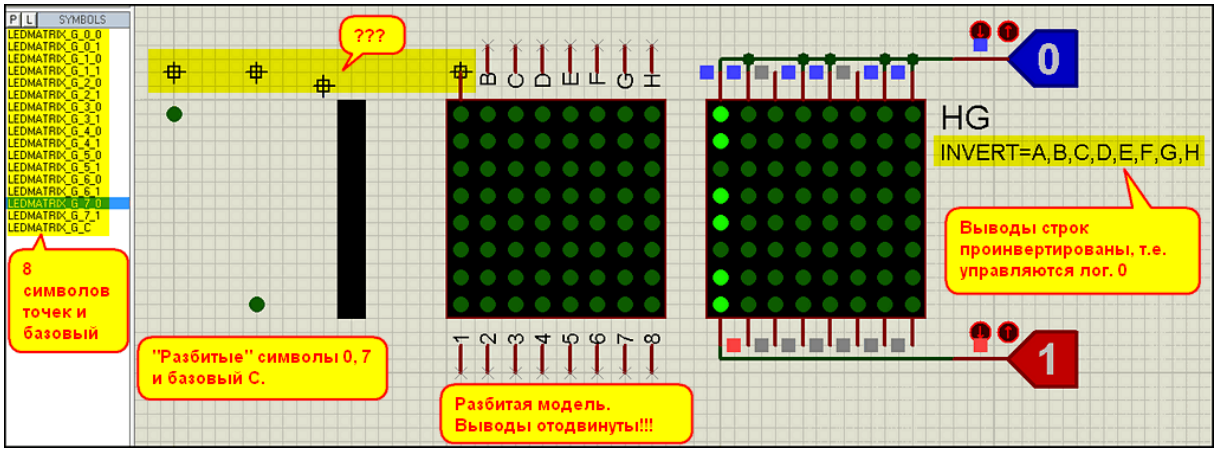


Рис. 87

Я пошел на небольшое ухищрение, чтобы не делать много скриншотов - совместил максимум в одном. Итак, на рисунке также представлены «декомпозированные» символы с индексами 0, 7 и базовый. И тут меня ожидал очень подозрительный сюрприз. Обратите внимание, что у символа **LEDMATRIX_G_C** после **Decompose** маркер **ORIGIN** сдвинут влево и ниже. Я оставил подсветку сетки с шагом **50th** на скриншоте, чтобы вы имели представление о том: куда и насколько. Честно говоря, мне непонятен этот ход разработчика модели, поэтому в своей я это применять не стану и размещу маркер как и обычно. Сам символ имеет **LEDMATRIX_G_C** форму прямоугольника для одного столбца, т.е. он как бы и определяет ширину знакоместа для одной колонки (в семисегментниках цифры). В середине рисунка 87 показана полная графика «разбитой» **MATRIX-8X8-GREEN**. Я опять отодвинул от тела выводы и подсветил их имена, за исключением вывода **A**, который оставлен на месте для лучшей ориентации. Ну и наконец справа на рисунке показана подключенная матрица 8x8. Мы видим, что сверху (со стороны сегментов-букв) для зажигания точки необходимо подать логический ноль, а снизу (со стороны знакомест-колонок) – логическую единицу. Это подверждается тем, что в свойствах матрицы прописано **INVERT=A,B,C,D,E,F,G,H**. Я убрал фигурные скобки «скрывающие» это свойство и оно видно на рисунке. Весь этот подготовительный этап в примере вложения папка **Step_1**.

Ну и теперь, зная особенности **LEDMPX**, сразу становится ясно – что нам предстоит сделать для того, чтобы получить модель формата 16x16 точек.

В первую очередь изменяем основную графическую модель, т.е. растягиваем черный квадрат подложку и расставляем на нем погашенные точки. Напомню еще раз, что здесь используются не символы из селектора с внедренным в них **ORIGIN**, а просто графические изображения соответсвующих элементов – квадрата, круга. Само-собой разумеется, что нам предстоит добавить еще по восемь выводов – сверху с именами букв латиницы – **I, J, K, L, M, N, O, P**, а снизу с именами в виде чисел – **9, 10, 11, 12, 13, 14, 15, 16** (Рис. 88).

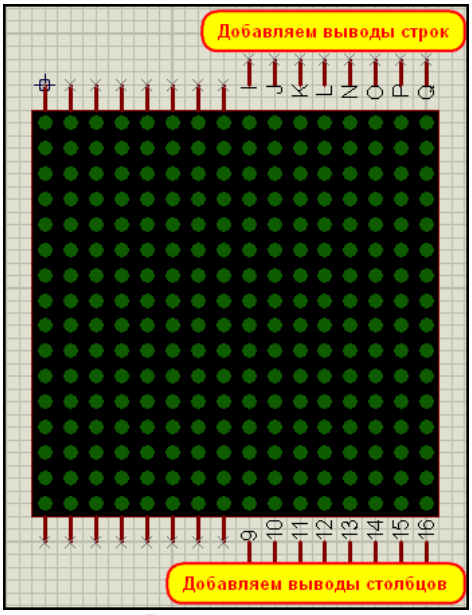


Рис. 88

Далее нам предстоит скомпилировать с помощью **Make Symbol** недостающие символы погашенных и светящихся точек в столбце и удлинить вниз прямоугольник базового символа. И вот здесь нас ожидает еще одна неприятная неожиданность Протеуса, о которой я «в пылу азарта» забыл упомянуть. Максимальное количество букв, цифр и спецсимволов подчеркивания в имени символа не должно превышать 15. Кстати, это же ограничение качается и **Device Name** на первой вкладке **Make Device**, так что возможно кто-то уже сталкивался с ним. Теперь посмотрим внимательно на исходные имена символов, начинающиеся с **LEDMATRIX_G**. Это уже 11 позиций, значит, нам под остальное осталось только 4. Делалась эта модель еще в ранних версиях Протеуса и тогда автор об этом не задумывался. Пока номера символов состояли из одной цифры, этого хватало, но, мы то пойдем дальше – будут и 10 и 11 и т.д. Ну что-ж, придется делать все символы сначала, укоротив имя. Я просто выбросил гласные буквы из слова MATRIX, и мои новые символы имеют обозначение **LEDMTRX_G**. Почему я не убрал вместо этого **_G** в конце имени? Да все из того же любопытства – не повлияет ли этот первый символ подчеркивания, находящийся в имени символа, на работу модели. Забегая вперед, скажу, что не повлиял. Следовательно, допустимо использовать в имени символа знак подчеркивания.

Но, продолжим наш титанический труд. Формирование новых символов, да и вообще работу с графикой активных индикаторов лучше делать с шагом сетки **50th**. И править их лучше в непосредственной близости от готового графического изображения всего компонента, чтобы не потерять ориентацию. Для начала удлиним базовый символ и скомпилируем его с новым именем **LEDMTRX_G_C** (Рис. 89). Попутно восстанавливаем «статус кво» маркера **ORIGIN** – я разместил его непосредственно над прямоугольником на том же расстоянии, что и в графическом изображении всей модели.

Рис. 89

Рис. 90

Затем начинаем формирование погашенных и светящихся точек матрицы. Поступаем также, ка и с базовым символом помещаем их поблизости от изображения всего компонента на уровне первой строки. **Make Symbol** из левой погашенной точки и маркера **ORIGIN** над ней символ **LEDMTRX_0_0**, а из светящейся точки и ее маркера **ORIGIN** – символ **LEDMTRX_0_1**. Затем смещаем обе точки (только точки, без маркера) на две клеточки вниз – вторая строка матрицы, и создаем соответствующие символы **LEDMTRX_1**, опять смещаем вниз – создаем **LEDMTRX**_3 и так до символов последней строки матрицы **LEDMTRX**_15.

После этого нам осталось только **Make Device** нашу новую модель, но не забудьте, что мы поменяли имена символов и увеличили их количество. Поэтому на первой вкладке **Make Device** изменяем эти значения (Рис. 91). Если при создании модели используется скрипт от старой модели 8x8, то в имени девайса достаточно просто поправить имя, если же вы создаете устройство «с нуля», то необходимо ввести **Device Name** полностью. Не забудьте также и про флажки бит-зависимости и связи с DLL.

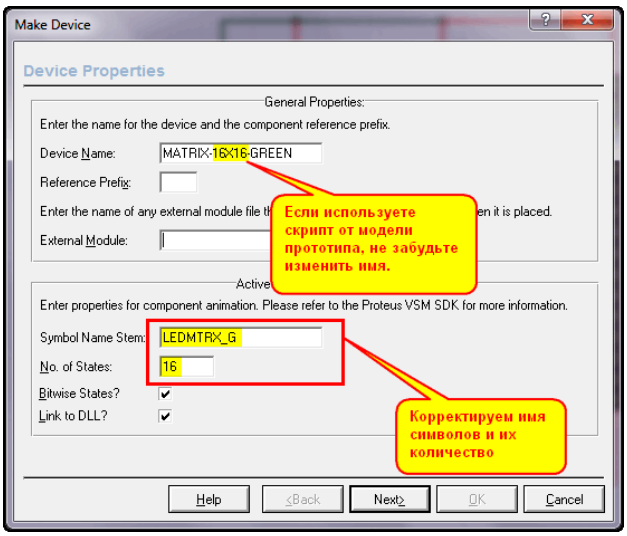


Рис. 91

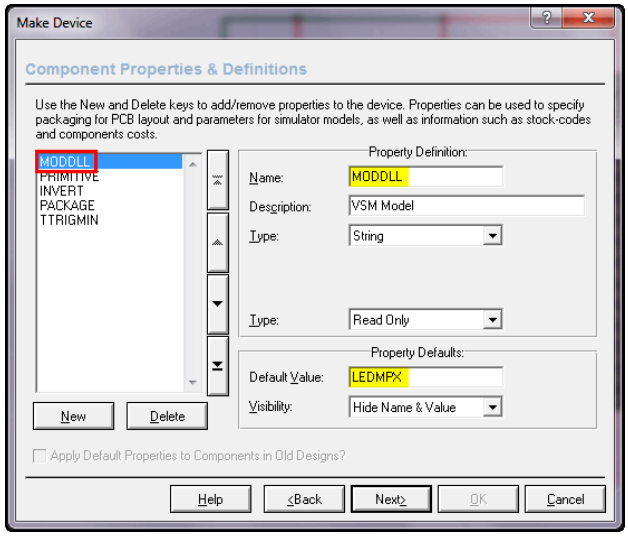


Рис. 92

Вторую вкладку назначения корпуса проходим транзитом, на третьей остановимся подробнее. Если использовался старый скрипт, то свойства будут взяты оттуда и здесь, кроме добавления имен сегментов (точек) в свойство **INVERT**, ничего менять не надо. Если же мы создаем устройство заново, то необходимо через кнопку **New** добавить и отредактировать следующие свойства:
MODDLL – В графе **Description** умолчание или по своему усмотрению, первый **Type** – **String**, второй **Type** – **Read Only**, **Default Value** – **LEDMPX**, или полностью **LEDMPX.DLL** (Рис. 92);
PRIMITIVE – В графе **Description** умолчание или по своему усмотрению, первый **Type** – **String**, второй **Type** – **Hidden**, **Default Value** – **DIGITAL,LEDMPX** – через запятую без пробелов (Рис. 93);
INVERT – В стандартных нет, добавляется через **Blank Item**. В графе **Description** умолчание или по своему усмотрению, первый **Type** – **String**, второй **Type** – **Hidden (**но тем, кто активно работает, с матрицами рекомендую поставить **Normal**, чтобы оперативно менять полярность выводов), **Default Value** – **A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P** – все через запятую без пробелов (Рис. 94);
TTRIGMIN – В стандартных нет, добавляется через **Blank Item**. В графе **Description** – **Minimum Trigger Time** (русскоязычные пользователи могут вписать – **Время переключения**), первый **Type** – **Float**, выбрать ограничения **Limits** – **Positive**, **Non-Zero**, второй **Type** – **Normal**, **Default Value** – **1ms** (Рис. 95).

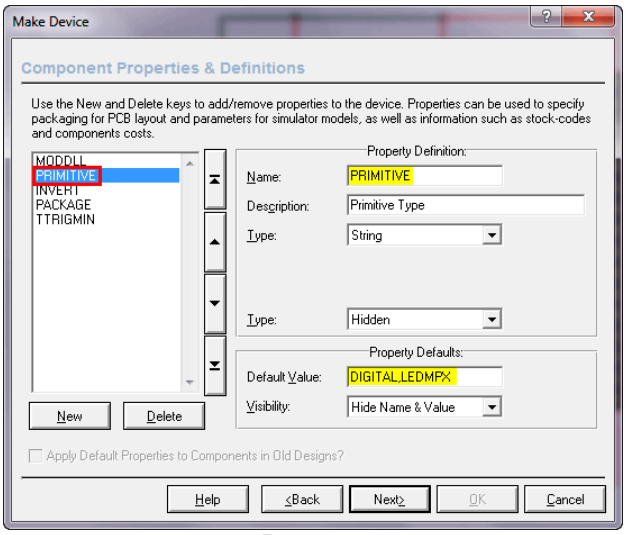


Рис. 93

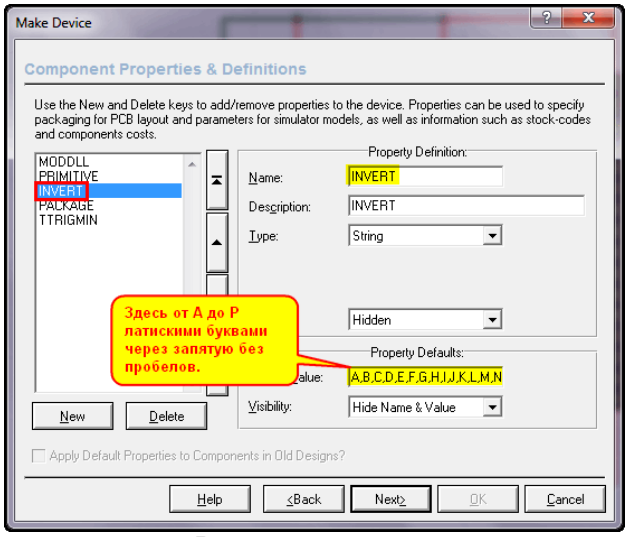


Рис. 94

Я ничего не сказал по **PACKAGE** – корпус, потому что пока мы его не добавляли и он по умолчанию примется **None** (отсутствует). Кроме того, для **TTRIGMIN** я указал значение, как и у остальных моделей – 1 миллисекунда, хотя сам же рекомендовал его уменьшать. Но это сделано только для того, чтобы не путаться при добавлении в проекты, а то у одной модели так, у другой по-другому.

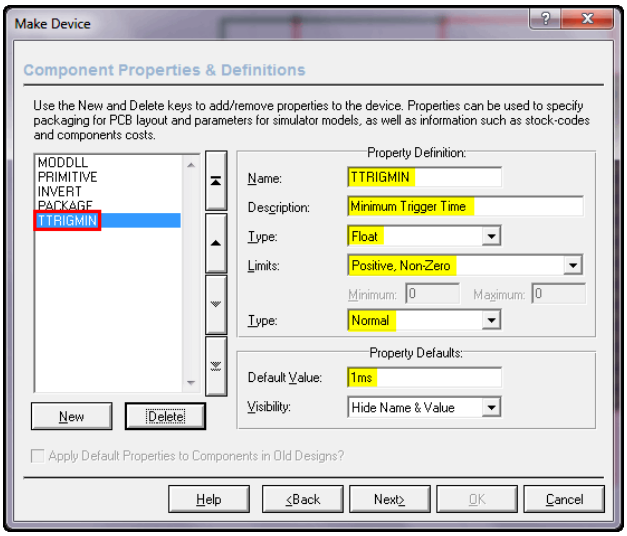


Рис. 95

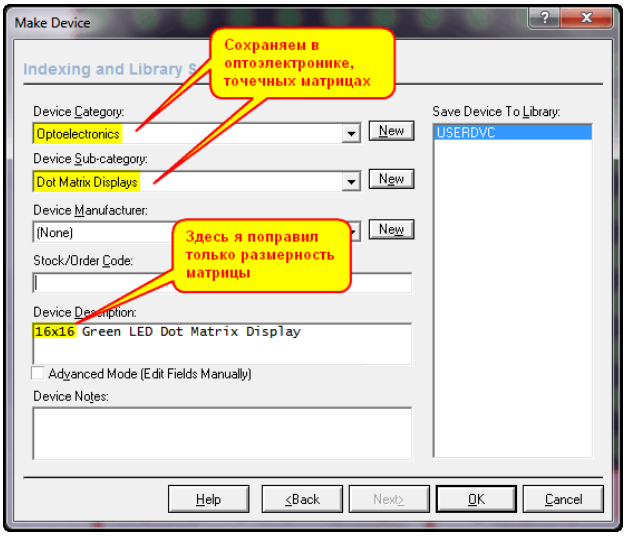


Рис. 96

Проходим нашу процедуру **Make Device** до конца и сохраняем матрицу в разделе оптоэлектроники, в подкатегории точечных матриц (Рис. 96). У меня для сохранения на данный момент открыта только **USRDVC**, но вы можете сохранить в своей библиотеке, предварительно создав ее в **Library Manager**. Напомню, что все остальные матрицы лежат в **DISPLAY**, но там только 13 свободных мест. Если собираетесь и далее создавать активные индикаторы, то лучше создать свою библиотеку, например, **DISPLAY2** на сотню «посадочных мест».

Нам осталось только проверить нашу матрицу в работе, что я и сделал (Рис. 97). Здесь я специально подсветил только угловые и центральные точки матрицы, чтобы убедиться, что нет повторных подсвечиваний точек и библиотека **LEDMPX.DLL** ведет себя адекватно с таким размерностью матрицы.

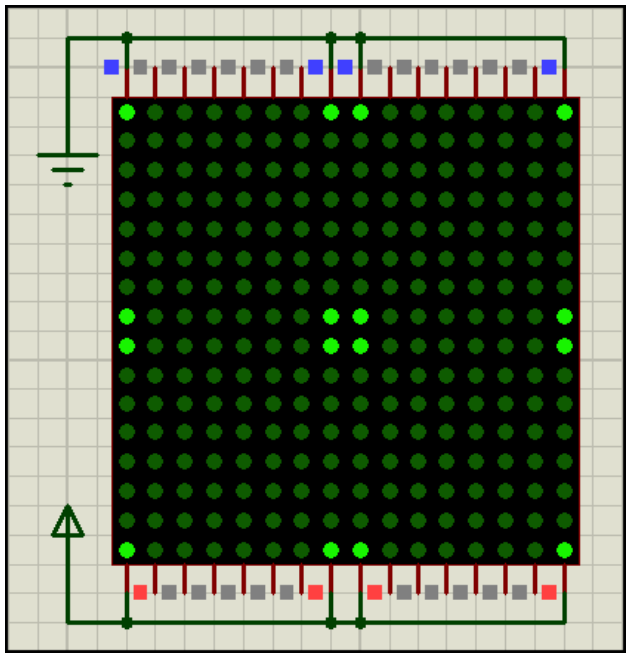


Рис. 97

На этом рассмотрение данной динамической библиотеки завершено, но не закончен материал по ее применению. Я вознамерился окончательно расставить точки над «и», да и в матрице тоже, с вопросами по динамической индикации в ISIS. Поэтому в следующем материале на примере данной матрицы мы этим и займемся. Первоначально хотел сделать это прямо здесь, но думаю, данный вопрос заслуживает отдельного параграфа.

К содержанию

8.9. О параметрах анимации и моделировании динамической индикации с помощью индикаторов на базе LEDMPX.DLL.

Я вновь возвращаюсь к теме отображения динамической индикации в ISIS, потому что теперь, когда мы знаем принцип работы многоразрядных индикаторов на основе **LEDMPX.DLL**, пора поставить огромный и жирную точку на этом вопросе. Для начала проведем несколько

экспериментов с моделью матрицы 16x16 из предыдущего раздела. Воспользуемся примитивами сдвиговых регистров **SHIFTREG_16** и соберем простейший «бегущий огонь» (Рис. 98).

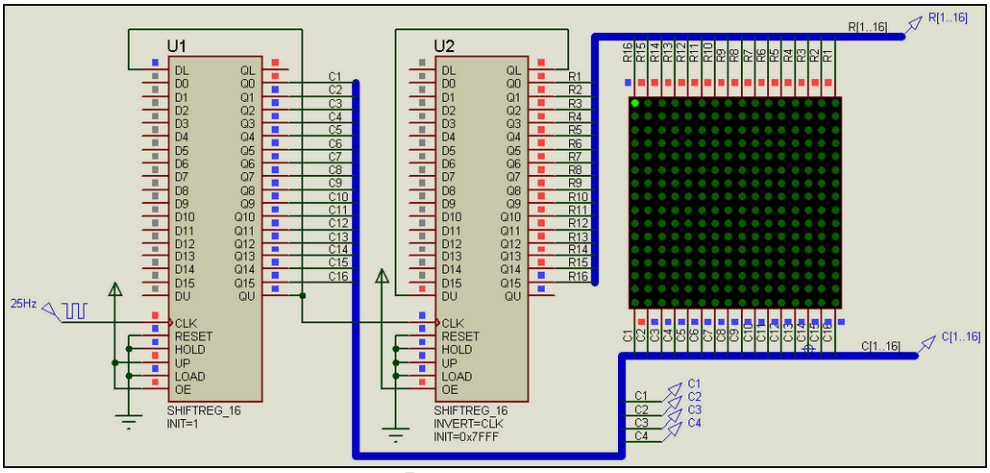


Рис. 98

В качестве тактового генератора используем **DCLOCK** из **Generator Mode**. При таком включении мы должны иметь светящуюся точку, скользящую слева направо по столбцам и постепенно переходящую сверху вниз по строкам (пример **Takt_25Hz.DSN** в папке **BAD_EXAMPLES\DOT_MATRIX** вложения).

Запустим симуляцию кнопкой **Step** или **Pause**, чтобы встать на нулевой отсчет времени. Этот момент зафиксирован на рисунке 98. В соответствии с прописанными в свойствах **INIT=1** для **U1** и **INIT=0x7FFF** для **U2** регистры приняли заданные значения, и загорелась верхняя левая точка индикатора. Выполним шаг кнопкой **Step**. Согласно **Single Step Time** (Рис. 99) наш симулятор продвинется на 50мс.

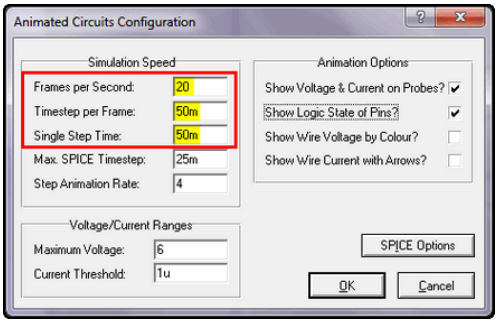


Рис. 99

При этом должна загореться вторая точка в строке, а первая погаснуть. Но тут нас ждет весьма неожиданный сюрприз (Рис. 100).

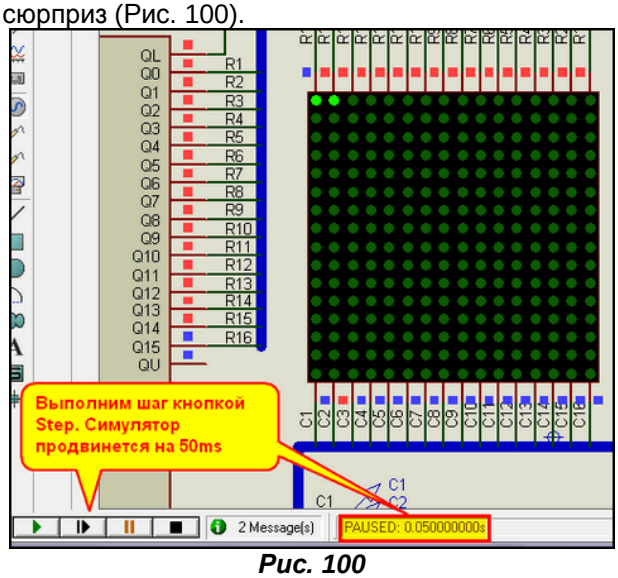


Рис. 100

По логическим уровням вроде все верно, низкий на строке **R1** и высокий на колонке **C2**, а горят сразу две точки вместо одной. Сколько бы мы далее не нажимали кнопку **Step**, время будет прибавляться по **50ms**, но светиться всегда будут две, а иногда даже три точки, т.е. мы имеем «размазанную» индикацию. Что же произошло – глюк Протеуса? Конечно глюк, но спровоцированный нами.

Для начала еще раз заглянем в меню **System => Set Animation Option** и вспомним, что означают и как соотнесены параметры, обведенные красной рамкой на Рис. 99.

Frames per Second (Кадры в секунду) – частота обновления окна программы при воспроизведении анимации (мультфильма, если так будет понятнее), запущенной кнопкой **Play**. Подчеркну, что этот параметр для окна основного поля **ISIS**, где расположена наша схема, а не всего экрана монитора. По умолчанию это значение равно **20Гц**, из чего несложно сделать расчет, что один фрейм соответствует **1000/20=50мс**. Данный параметр в **ISIS** можно менять в пределах от 10 до 50мс, дальше ограничения ISIS не позволят этого сделать.

Timestep per Frame – временной интервал анимации, приходящийся на один фрейм. Обратите внимание, что если мы разделим все те же **1000мс** (1секунда) на значение этого параметра по умолчанию **50мс**, то получим частоту **20Гц**, ту же, что и в предыдущем параметре.

Single Step Time – временной интервал анимации, приходящийся на один шаг, выполняемый по кнопке **Step**. По умолчанию он установлен равным предыдущему параметру, но при желании можно поставить и отличающееся значение.

Также хочу напомнить, что установленные параметры анимации **сохраняются в текущем проекте**, так что если вы сохранили проект и открыли новый, то там они снова встанут в значения по умолчанию, т.е. соответственно **20Гц**, **50мс** и **50мс**. Это соотношение 20 кадров в секунду и 50мс выбрано неспроста. Как гласит **ProSPICE Help**, оно обеспечивает наиболее гладкое зрительное восприятие симуляции в реальном времени в большинстве случаев и при этом не перегружает ЦП компьютера дополнительными расчетами активных элементов в кадре.

Теперь создадим цифровой график нашей индикации и попробуем разобраться по нему в причинах глюка. На этот график (Рис. 101) для наглядности я нанесу границы наших фреймов (кадров) через каждые 50мс красными вертикальными линиями. Кроме того, с помощью дополнительного генератора я имитировал время стробирования нашего индикатора **Trigger Time**, равное по умолчанию 1мс.

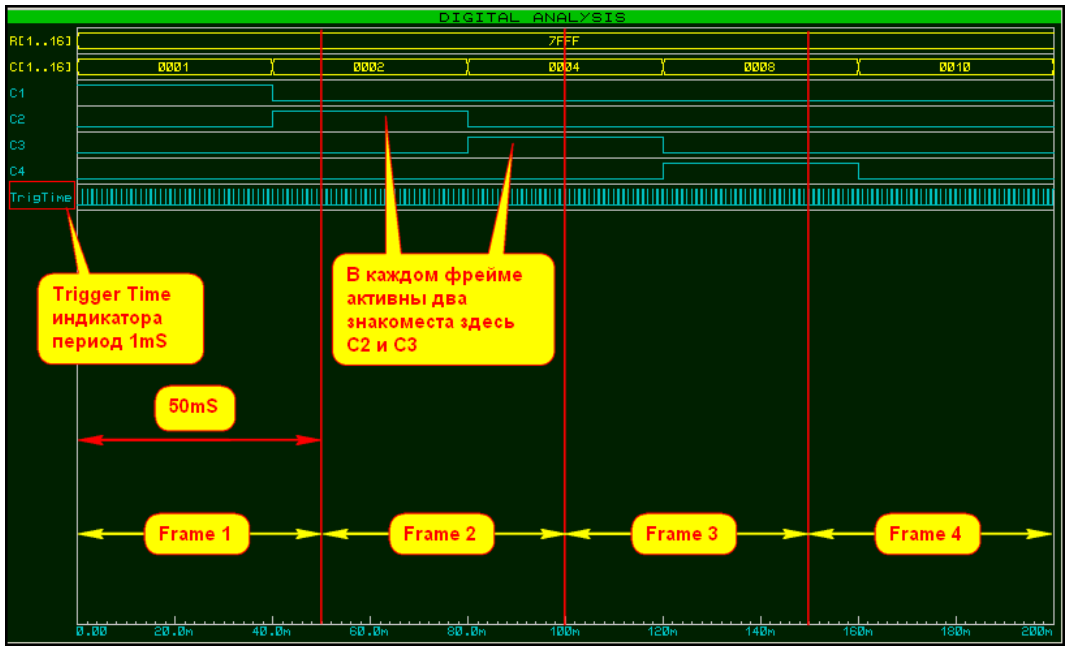


Рис. 101

Обратите внимание на первые три фрейма. В каждом фрейме мы имеем по два активных сигнала (лог. 1) знакомест-колонок **C1…C4**. Поскольку информация в строке не меняется, она имеет одно и то же значение для всех кадров , относящихся к первой строке. Модель **LEDMPX.DLL** стробирует активность элемента с периодичностью **1ms**, т.е. в кадре (фрейме) длительностью **50ms** опрос выполняется 50 раз, и, если встречается совпадение лог. 0 на строке матрицы и лог. 1 на знакоместе-колонке – зажигается соответствующая точка. У нас бы и при старте две точки загорелись, но мы стояли на временном отсчете 0, и поэтому **LEDMPX** просто еще не успела «отработать» на второе знакоместо **C2**.

Именно такая же чехарда с «наползанием» разрядов-знакомест друг на друга творится, когда мы запускаем проекты с динамической индикацией на сегментных индикаторах, если не

принять превентивных мер. Выражаясь словами телевизионщиков, мы имеем «сбитую кадровую синхронизацию». Если мы увеличим тактовую частоту задающего генератора (соответствующие примеры в папке **BAD_SAMPLES\DOT_MATRIX** вложения), то в кадр будут попадать уже не два разряда, а больше и в конечном итоге (пример с генератором 100 kHz) мы получим уже на втором шаге симуляции полностью светящуюся матрицу. Это одна из причин неадекватной динамической индикации в ISIS.

Прежде, чем мы пойдем дальше - полезный совет владельцам «медленных компьютеров», подтвержденный примерами в папке **BAD_SAMPLES\DOT_MATRIX** вложения.

СОВЕТ: *Обратите внимание на то, что часть примеров в папке **BAD_SAMPLES** имеет на конце индекс **Lite** – облегченные. Еще раз вернемся к рисунку 99, а конкретно к двум установленным флажкам в правой части свойств анимации. Первый из них – **Show Voltage & Current on Probes?** – отвечает за показ значений напряжений и токов у установленных в проекте пробников-зондов. Второй – **Show Logic State on Pins?** – отвечает за показ тех самых сине-красных квадратиков логического состояния выводов у цифровых моделей. Так как с некоторого времени мне приходится часть материала готовить на нетбуке с тактовой 1,6 ГГц и слабенькой видеокартой, то даже некоторые примеры с чисто цифровой симуляцией грузят бедного «нетбуку» на все 100%. Так вот, убрав эти флажки, я снижаю загрузку ЦП до приемлемого уровня. Это заметно и на мощных компьютерах, можете убедиться самостоятельно, запустив одноименные примеры из папки с индексом **Lite** и без такового. Примите на вооружение данный прием, хотя он наиболее эффективен только при большом количестве пробников и «многоногих» цифровых моделей в проекте.*

Итак, для того, чтобы привести рассмотренный выше пример в норму, возникает естественное желание покрутить «ручку частоты кадров», чтобы границы кадров совпали с фронтами смены знакомест-колонок **C1…C4**. В данном случае мы можем позволить себе такую роскошь. При заданной нами для входного генератора частоте 25Гц время свечения (длительность единичного импульса) на входах столбцов индикатора **Cx** составляет 40 мс. Зададим для **Timestep per Frame** и **Single Step Time** это значение. Чтобы сохранить соотношение между **Timestep per Frame** и **Frames per Second** пересчитаем последнее **1000/40=25** кадров в секунду. Для сохранения эффекта реального времени задаем **Frames per Second** найденное значение 25, пока мы еще не выходим за рамки заданных ограничений от 10 до 50. Запускаем симуляцию, но нужного эффекта не получаем, по-прежнему светятся по две точки. Можно и дальше изменять значения этих параметров, но эффекта это не даст. В конечном итоге мы упремся в верхнее ограничение 50 для параметра кадров в секунду, и дальнейшее снижение таймстепов приведет только к замедлению смены картинок анимации. Правда, справедливости ради надо отметить, что нагрузка на ЦП компьютера при этом начнет снижаться, так что в ряде случаев – это тоже полезно. Снизится загрузка и замедлится мультфильм если мы будем просто менять один из параметров, например, уменьшать время **Timestep per Frame**, не затрагивая при этом частоту кадров в секунду.

Кроме того, сейчас мы экспериментируем с одним светящимся объектом-точкой, а представьте, что у нас сегментный индикатор на несколько знакомест, где надо видеть одновременно все разряды. Это уменьшение **Timestep per Frame** до границ одного разряда даст нам мелькающие по очереди отдельные разряды и испортит целостность восприятия нашего «Ну, погоди». Вероятно, многие уже сталкивались с таким вариантом, а для тех, кто желает полюбоваться на примеры - вложение в папке **BAD_EXAMPLES\MC** соответственно для AVR с кодом на Си в CodeVision и для PIC с кодом в CCS PICC. Конечно же, для обеспечения стабильного периода смена разрядов индикации загнана в прерывание по переполнению таймера 0, а частота выбрана заведомо низкой, чтобы можно было «поиграть» с параметрами анимации в ISIS. На основании этих примеров уже можно сделать важные выводы.

Для того, чтобы при многоразрядной динамической индикации получить реальную картинку необходимо, чтобы кадр анимации в симуляторе захватывал полный цикл индикации, т.е. длительность кадра должна быть не менее суммы длительностей индикации всех разрядов.

Ну и второй, не менее важный момент, который неоднократно рассматривался на форуме и до меня – наиболее простой метод получить стабильный цикл индикации, это смена разрядов в прерывании по переполнению одного из таймеров микроконтроллера. Именно нестабильность периода индикации, который зависит от подаваемой на вход частотомера Денисова измеряемой частоты, приводила к срыву картинки в рассмотренном в самом начале FAQ примере. Там мы побороли этот глюк другим методом, которого коснемся чуть ниже. Кстати, если кто-то захочет воспользоваться кодом из примеров индикации в реальном устройстве, хочу остановиться еще на одном моменте. Я делал все исключительно для примера и соответственно упрощенно – только индикация. Поэтому у меня отсутствует запрет на прерывания в начале обработчика и разрешение в конце. В реальном девайсе это может сыграть злую шутку, если в момент обработки прерывания по таймеру прилетит еще одно с более высоким приоритетом. Об этом тоже нелишне помнить.

Ну а пока вернемся к примеру с **DOTMATRIX** индикатором и проанализируем – почему по-прежнему светятся две точки? Вот тут и выходит на первый план параметр **Minimum Trigger Time** модели на основе **LEDMPX.DLL**. Попробую объяснить это графически, поскольку чисто словесное

описание этого момента, как я не пытался, получается слишком запутанным для понимания. Обратимся к рисунку 102. Здесь я условно принял, что граница фреймов **Frame1** (бирюзовый фон) и **Frame2** (бледно-зеленый фон) совпадают с фронтами смены сигналов по строкам **R1** и **R2** и по столбцам – **C1** и **C2**, т.е. выполняется условие рассмотренное выше.

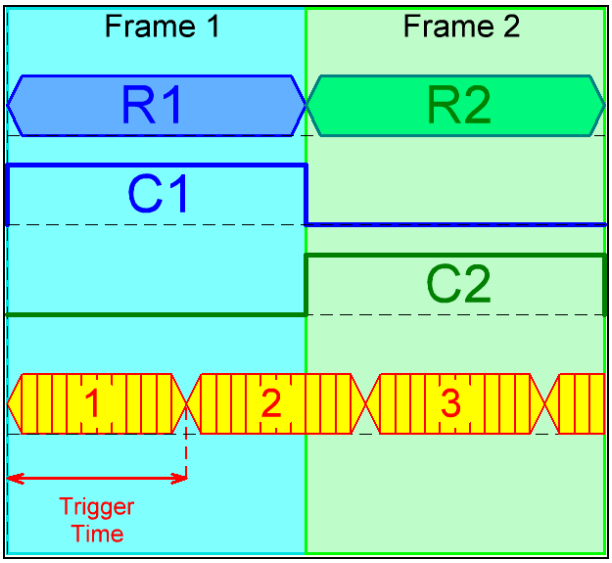


Рис. 102

По умолчанию **Minimum Trigger Time** для нашего индикатора задано равным 1мс, т.е. для одного кадра длительностью 50мс **LEDMPX.DLL** рассчитывает состояние выводов индикатора 50 раз. Причем нет никакой гарантии, что границы стробирования **Minimum Trigger Time** будут четко совпадать с границами кадров и смены знакомест и очередной строб не заползет большей своей частью в соседний разряд (знакоместо). Я даже допускаю мысль, что внутри интервала Trigger Time программист заложил усреднение нескольких рассчитанных значений, поэтому и нарисовал дополнительные красные стробы. Если бы я писал модель, то вероятно так бы и сделал. Но в реальности очень похоже на то, что «первую скрипку» тут играет конечное значение Trigger Time, т.е. как бы задний фронт этого интервала. Как мы помним, на основании рассчитанных данных **LEDMPX** принимает решение о том, какие активные элементы должны светиться в данном кадре. А теперь посмотрим на интервал обозначенный цифрой 2. Какое решение выдаст **LEDMPX** для этого интервала? Он вроде бы относится к первому кадру, но цепляет данные и для **R2** и **C2**. Естественно, это вызовет подсвечивание в данном кадре активных элементов соседнего знакоместа. А теперь применим гашение на границе смены разрядов, ну и кадров естественно. Для этого просто достаточно убрать данные, например на шине **R** на интервале длительностью чуть больше нашего **Minimum Trigger Time** (Рис. 103).

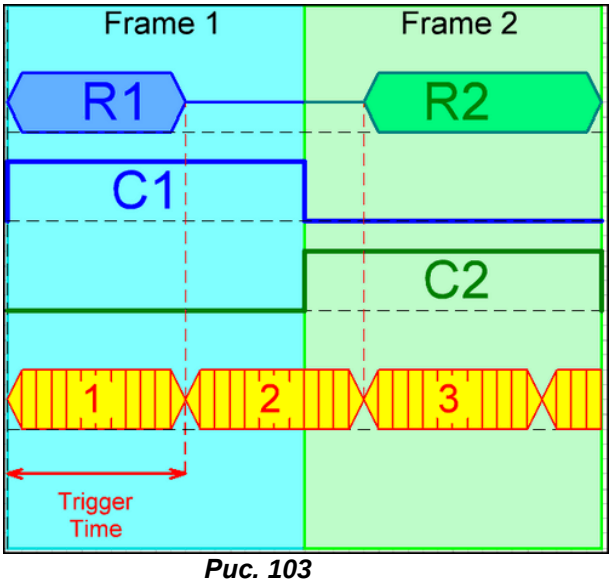


Рис. 103

Вот и вся операция. Интервал 2, хоть и заползает на второй кадр, но засветки данных из соседнего знакоместа не даст – для него отсутствуют активные сигналы в строке R, а следующий

уже полностью относится ко второму кадру. Причем, как мы убедились на примере частотомера в начале FAQ, такой прием работает и при нестабильном периоде индикации. Большого умственного напряжения это не требует. Например, в листингах на Си примеров для микроконтроллеров во вложении к этому разделу вначале обработчика прерывания стоит закомментированной всего одна строка, которая убирает сигнал с порта, управляющего сегментами на момент переключения порта знакомест. Вот она то и даст этот эффект.
Ну а сейчас мы обойдем этот момент другим путем. Правда, сразу оговорюсь, что этот прием работает только при стабильном периоде индикации. Давайте подумаем, а надо ли стробировать данные 50 раз за период индикации, если они при этом не меняются? Возьмем, да и приравняем **Minimum Trigger Time** к длительности активного сигнала для знакоместа. Вот именно равное значение в данном случае не работает, но стоит чуть-чуть увеличить **Minimum Trigger Time**, например, длительность сигнала знакоместа равна 10мс, а значение **Trigger Time** мы установим равным 10,001мс, и картинка приходит в норму. Попробую объяснить, почему такое происходит. При равных значениях видимо по-прежнему сказывается захват соседнего знакоместа. Если же **Minimum Trigger Time** чуть больше длительности знакоместа, сбой тоже происходит, но совпадение фазы сигналов настолько редко, что на общем фоне большого количества кадров этот сбой практически не мешает зрительному восприятию картинки. Иногда мелькнет неадекватная информация, и опять нормальное отображение. Причем, еще раз подчеркну, в данном случае речь идет не о полном цикле индикации, а именно о длительности одного знакоместа. Примеры использования данного приема в папке вложения **GOOD_EXAMPLES**. В случае с активной матрицей мы при равных длительности кадра и сигнале столбца со значением **Minimum Trigger Time** чуть большим, чем сигнал столбца получаем единственную светящуюся точку как в шаге, так и при запущенной кнопкой **Play** симуляции. В случае многоразрядного сегментного индикатора если долго и внимательно смотреть на индикатор, то можно засечь проскок наложения разрядов, но общей картине восприятия он не мешает.
Ну и чтобы окончательно покончить с этим вопросом, подытожим вышесказанное.

Для симуляции динамической индикации в ISIS в любом случае наиболее предпочтителен способ гашения индикатора на момент смены разрядов (знакомест). При этом длительность интервала гашения должна быть хотя бы чуть больше длительности параметра **Minimum Trigger Time**, который установлен для индикатора.

При стабильном периоде индикации можно попробовать добиться желаемого эффекта, установив параметр **Minimum Trigger Time** чуть больше длительности активного сигнала одного знакоместа.

Еще один важный момент, который я чуть не упустил. В своих примерах я использовал индикацию с минимальной частотой, чтобы можно было отследить изменение параметров анимации. На деле же динамическую индикацию стараются сделать с частотой выше 50 Гц, чтобы исключить мерцание реальных индикаторов. Естественно, подогнать параметры анимации ISIS под один период индикации в таких случаях нам не удастся из-за программных ограничений Протеуса, но задать длительность кадра таким, чтобы в него входило некоторое целое количество периодов можно и даже желательно. Для этого достаточно исследовать индикацию с помощью цифрового графика, как это сделано в приведенных примерах. Дальше все построено на банальной арифметике.

Ну и в заключение несколько слов о том, каким выбрать **Minimum Trigger Time**. На него, в отличие от параметров анимации, нет жестких ограничений. Тут тоже все зависит от конкретного построения схемы и программы динамической индикации. В большинстве случаев все работает и с параметрами по умолчанию, но иногда приходится повозиться. В сторону уменьшения значение ограничено лишь тем, что при маленьком значении на индикаторе начнут светиться данные соседних знакомест (Рис. 104). В сторону увеличения **Minimum Trigger Time** ограничено длительностью кадра анимации и если превысит его, то у нас просто начнутся пропуски данных – хаотичное моргание отдельных разрядов (Рис. 105), или индикация будет отсутствовать. Если применяется гашение при смене разрядов, то значение **Minimum Trigger Time** надо установить меньше, чем интервал гашения.

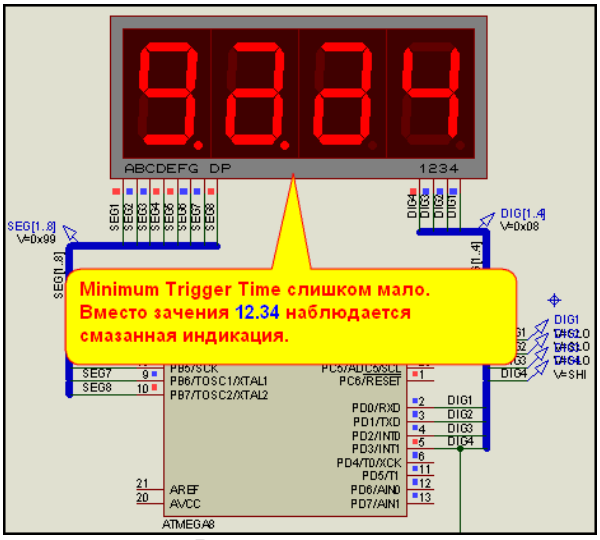


Рис. 104

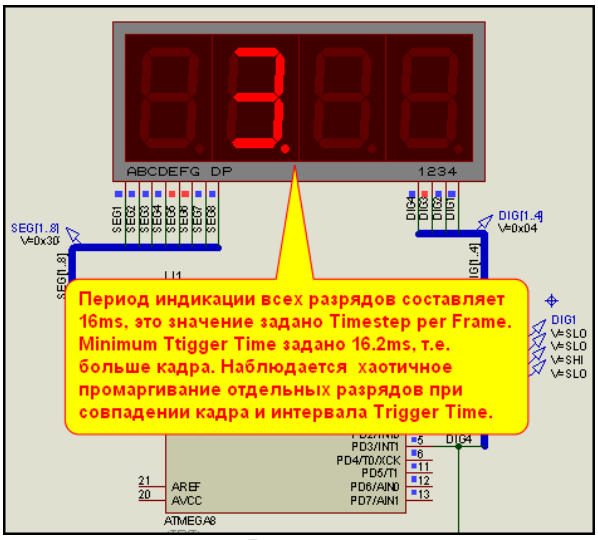


Рис. 105

Эти примеры расположены в папке **MC_TR_TIME** вложения. Соответствующие пояснения даны в комментариях на поле проектов. Ну а я завершаю окончательно и бесповоротно материал о динамической индикации в Протеусе и надеюсь больше к этому вопросу не возвращаться. Теперь вы знаете все о поведении моделей **LEDMPX** при симуляции, а выбор метода реализации динамической индикации остается за вами. Ну а мы переходим к не менее интересному материалу, который уже давно заждался, – индикаторы на базе **LCDMPX.DLL**.

К содержанию

8.10. Знакомимся с моделями на основе LCDMPX.DLL – еще одним вариантом библиотеки для построения цифровых индикаторов в ISIS. Общие принципы построения моделей ЖК индикаторов.

Из названия библиотеки по аналогии с описанной выше напрашивается элементарный вывод, что служит она для создания моделей жидкокристаллических индикаторов (**LCD** от английского **L**iquid **C**rystal **D**isplay), и так же как предыдущая мультиплексирована (**MPX** от multiplexing**)**. Ну, если второй термин себя полностью оправдывает, то **LCD** в общем то только потому, что она изначально разрабатывалась для имитации ЖК цифровых индикаторов. Но, как мы уже убедились на примере **LEDMPX**, в которой диодными свойствами и близко не пахнет, аналогично обстоит дело и здесь. **LCDMPX** тоже обладает только цифровыми свойствами, т.е. тока не «кушает» и имеет только два состояния: активный элемент «горит», активный элемент потушен. Я не стану расписывать здесь теорию управления реальными ЖК индикаторами, для этого есть соответствующая литература. Про особенности конструкций ЖК индикаторов и принципы частотного и фазового управления ими вы можете прочесть в старых справочниках, например:

- **МРБ вып. 1122. Пароль Н.В., Кайдаров С.А. Знакосинтезирующие индикаторы и их применение. М., Радио и связь, 1988 г.;**
- **Вуколов Н.И., Михайлов А.Н.. Знакосинтезирующие индикаторы. Справочник. Под ред. В.П. Балашова. М., Радио и связь, 1987 г.**

Особенности управления конкретными реальными ЖК индикаторами необходимо учитывать при разработке своих конструкций, поскольку от этого зависит «продолжительность жизни» вашего индикатора, а модель ISIS построена так, что допускает статическое управление и легко пропустит ошибки, допущенные в динамике. Еще раз хочу подчеркнуть, что пока мы ведем речь о простых индикаторах, у которых имеется один общий вывод для каждой группы сегментов индикатора, просьба «чайников» от электроники не путать с COG-индикаторами, их мы коснемся особо.

В библиотеках ISIS цифровые ЖК индикаторы представлены весьма скромно всего пятью моделями, расположенными в **Optoelextronics => LCD Panels Displays**. Объясняется этот факт по-видимому отсутствием спроса на данный тип моделей у легальных «забугорных» пользователей. Нам же, сирым и убогим, выковыривающим ЖК индикаторы из отслуживших свой срок калькуляторов, телефонов и прочей оргтехники отсутствие этих моделей доставляет массу неудобств. Да и китайско-белорусская промышленность до сих пор исправно снабжает Российский рынок сегментными ЖК-индикаторами. Это и широко распространенные индикаторы Минского **ОАО "ИНТЕГРАЛ"** http://www.integral.by и цифровые ЖК индикаторы китайской фирмы **Intech LCD Group** http://www.intech-lcd.com/standardpanel.htm. Подчас применение этих сравнительно дешевых индикаторов позволяет весьма существенно сэкономить деньги в простых приложениях типа электронных часов, таймеров, термометров и пр. «бытовухи». Ну и конечно, многим хочется иметь такие модели для предварительной отладки в ISIS. Вот этим мы сейчас и займемся.

Итак, для того чтобы активировать («зажечь») сегменты ЖК индикаторов на базе **LCDMPX** достаточно подать на них питание (пример индикатора **VI-402-DP** слева) или логические уровни (пример индикатора **VI-332-DP** справа) в правильной полярности (Рис. 106).

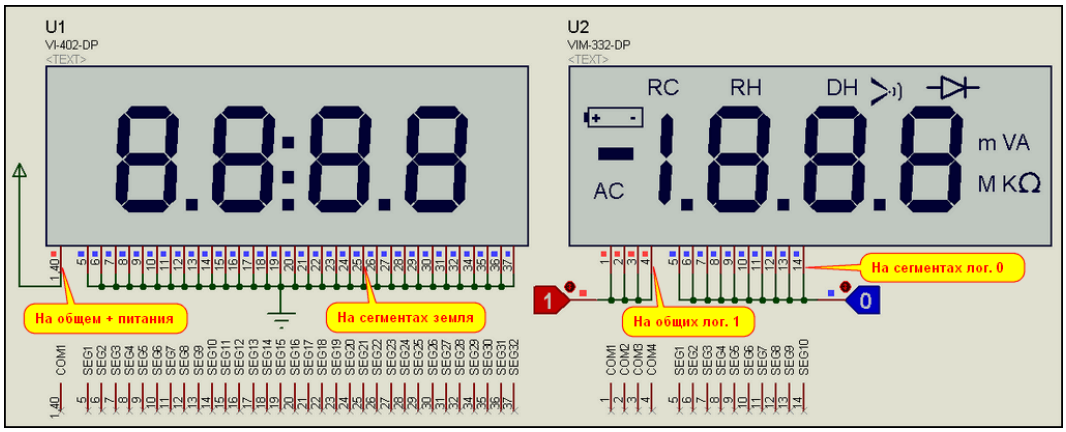


Рис. 106

По сути, этими двумя индикаторами: для часов и для мультиметра и ограничен наш выбор в ISIS. Оставшиеся три в библиотеке – заказные индикаторы для отладочных микроконтроллерных комплектов и практической ценности для рядового пользователя не представляют.
Чуть ниже индикаторов на рисунке я расположил их выводы с подсвеченными именами. Мы можем заметить, что часовой индикатор имеет один общий вывод с именем **COM1** и 32 сегментных вывода с именами **SEG1…SEG32**. Индикатор справа в отличие от первого имеет четыре общих вывода – **COM1…COM4** и меньшее количество сегментных. Наталкивает на мысль об аналогии с моделями на основе **LEDMPX**, рассмотренными ранее, но это не совсем корректно. Дело в том, что в модели **LEDMPX**, как вы вероятно помните, развертывание знакомест (нумерация) идет слева направо по горизонтали, т.е. общему выводу **1** соответствует левая цифра, выводу 2 вторая слева и далее с шагом шириной в одно знакоместо (цифру из N-сегментов с подложкой-фоном). В модели **LCDMPX** имеет место относительная двумерная пространственная ориентация по осям X-Y для сегментов, соответствующих определенному общему выводу с именем **COM**. На примере **VI-332-DP** для **COM1** и **COM4** расположение соответствующих сегментов представлено на рисунке 107. Не очень привычно, но дает более широкие возможности при создании собственных моделей.

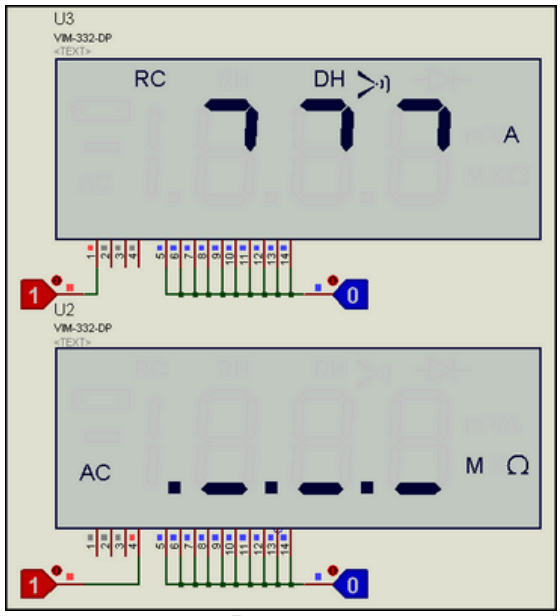


Рис. 107

Чтобы нам в дальнейшем не путаться выводы «знакомест» (конструктивно это выводы задней пластины ЖК индикатора) **COM**n от английского «common» – общий, я так и буду называть – общие выводы, потому что термин «знакоместо» здесь весьма условен, ну а выводы сегментов – **SEG**n так и останутся выводами сегментов. Итак, максимальное количество и тех и других для моделей на основе **LCDMPX** – 64. Нумерацию можно вести кому как привычнее, т.е. могут существовать **SEG1…SEG64** или **SEG0…SEG63**. Чем ограничение в 64 сегмента чревато в

действительности? Среди реально существующих цифровых ЖК индикаторов немного найдется с раздельными общими выводами. Как правило, он один на всю заднюю пластину-подложку. А вот выводов сегментов может оказаться больше, чем допустимый предел. Типичные примеры: таксофонный **ИЖЦ13-8/7** (8 цифр с точками и 8 символов надчеркивания над ними) или **Intech ITS-E0806** (8 цифр с точками плюс два разделительных двоеточия).

Дальнейшее рассмотрение продолжим на примере **VI-332-DP**, поскольку он наиболее показателен с точки зрения построения моделей на базе **LCDMPX.DLL**. Для начала «поколотим его молотком» (**Decompose**) и посмотрим на состав входящих в него символов, переключив селектор в соответствующее положение кнопкой **S** слева (Рис. 108).

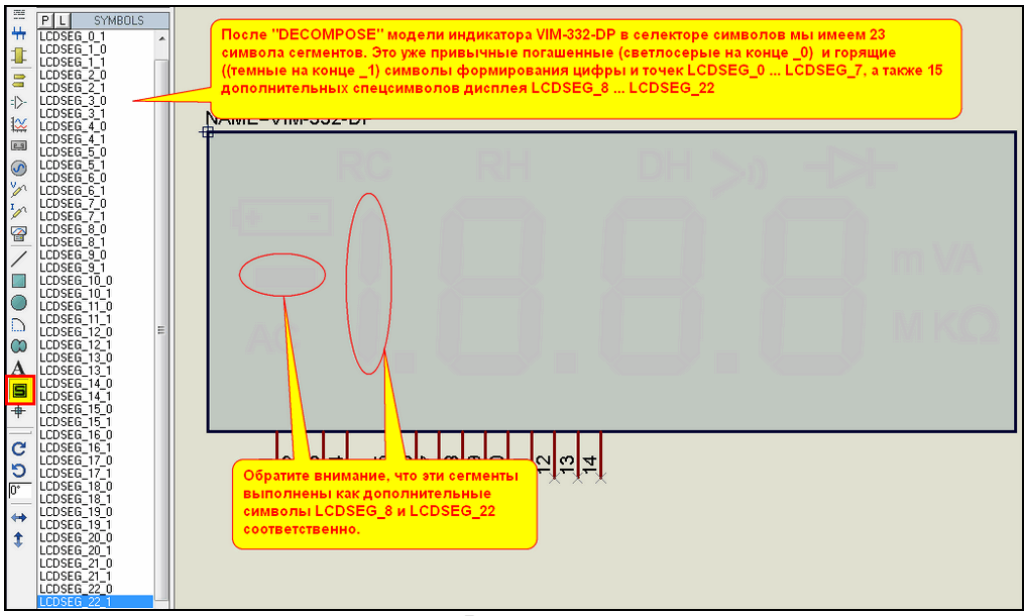


Рис. 108

Всего в составе индикатора 23 символа. Символы **LCDSEG_0**…**LCDSEG_7** нам уже знакомы по предыдущим семисегментным индикаторам. Это образующие цифру сегменты и десятичная точка. Для примера на рисунке 109 в верхнем ряду приведены первые четыре символа сегментов дополнительно «разобранные на запчасти», чтобы был виден маркер **ORIGIN**. Здесь тоже никаких новшеств по отношению к остальным сегментным индикаторам за исключением цвета погашенного (с индексом _0) и горящего (с индексом _1) символа.

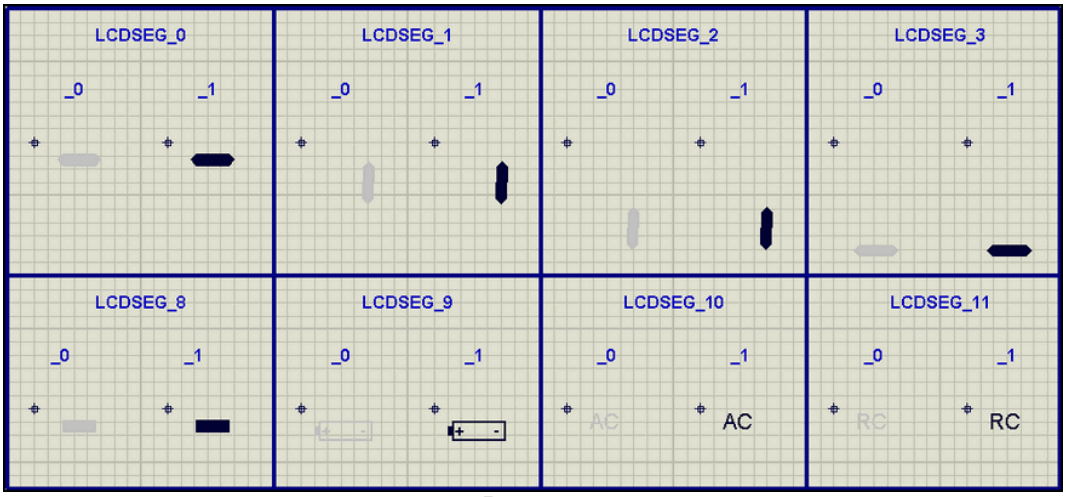


Рис. 109

В нижнем ряду также «разобранные на запчасти» первые четыре дополнительных символа. Здесь я хотел бы заострить ваше внимание на положении маркера **ORIGIN**, потому что именно с ним будет связано все, что будет изложено ниже, а именно принцип позиционирования символов в моделях на основе **LCDMPX.DLL**.

Снова вернемся к полноценной модели **VI-332-DP** и заглянем в ее свойства. В окне **Edit Properties** установим флажок **Edit all properties as text**, чтобы увидеть то, что изначально скрыто от посторонних глаз (Рис. 110).

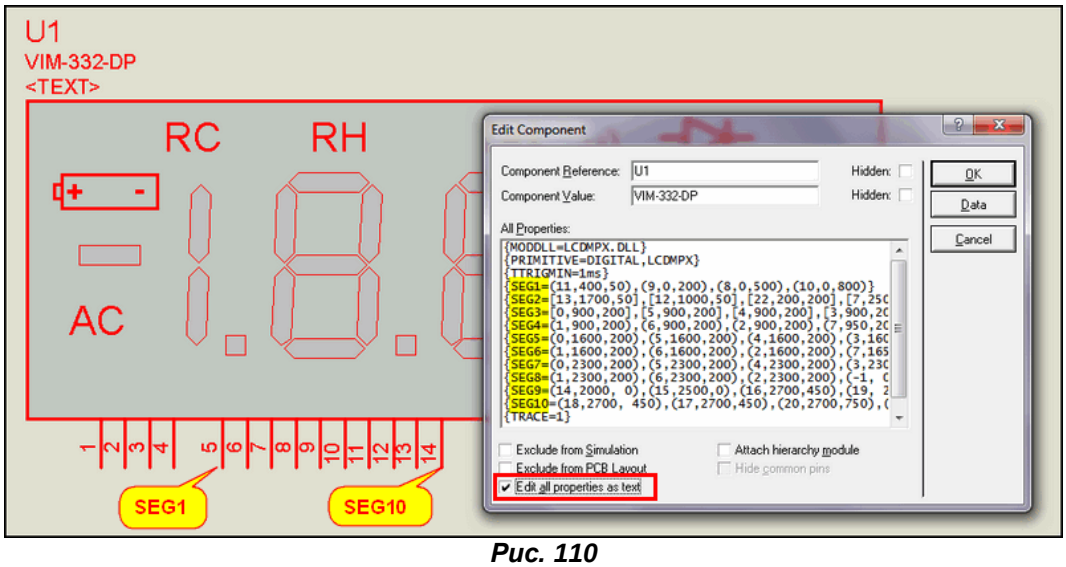


Рис. 110

Ниже я полностью привожу те десять строчек (по числу сегментных выводов SEG1…SEG10), которые нас интересуют:

```
{SEG1=(11,400,50),(9,0,200),(8,0,500),(10,0,800)}
{SEG2=[13,1700,50],[12,1000,50],[22,200,200],[7,250,200]}
{SEG3=[0,900,200],[5,900,200],[4,900,200],[3,900,200]}
{SEG4=(1,900,200),(6,900,200),(2,900,200),(7,950,200}
{SEG5=(0,1600,200),(5,1600,200),(4,1600,200),(3,1600,200)}
{SEG6=(1,1600,200),(6,1600,200),(2,1600,200),(7,1650,200)}
{SEG7=(0,2300,200),(5,2300,200),(4,2300,200),(3,2300,200)}
{SEG8=(1,2300,200),(6,2300,200),(2,2300,200),(-1, 0,0)}
{SEG9=(14,2000, 0),(15,2500,0),(16,2700,450),(19, 2700,750)}
{SEG10=(18,2700, 450),(17,2700,450),(20,2700,750),(21, 2700,750)}
```

Вот это и есть основа позиционирования сегментов и с ней нам сейчас предстоит разобраться. Итак, для начала вспомним, что фигурные скобки в начале и конце каждой строки означают сокрытие текста (**Hidden Text**) и поэтому нас в данный момент мало интересуют. Как вы наверное уже догадались, каждая строка соответствует выводу сегментов, с имени которого она начинается, т.е. **SEG1** соответствует одноименному выводу с номером 5, ну а **SEG10** – выводу с номером 14. После знака равенства имеются четыре секции по три числа, заключенные в круглые скобки и разделенные запятыми. Именно четыре общих вывода **COM** имеет данная модель, если кто подзабыл – вернитесь к рисунку 106. Нетрудно догадаться, что левая секция в круглых скобках соответствует выводу с именем **COM1**, следующая **COM2** и т.д. Нам осталось разобраться с тем, что находится внутри круглых скобок. Тут тоже ничего сверхъестественного: первое число – номер **n** символа **LCDSEG_n**, второе и третье соответственно положение по осям **X** и **Y** в координатах сетки маркера **ORIGIN** этого символа относительно маркера **ORIGIN** всей модели (виден в левом верхнем углу на Рис. 108). Вот именно это я и обозвал выше по тексту заумной фразой двумерная относительная пространственная ориентация. Чтобы было более наглядно, попробую изобразить это на картинке на примере одного символа, например, **LCDSEG_9** (Рис. 109) с координатами **X=0**, **Y=200**, соответствующего изображению батарейки. Данный символ в соответствии со второй секцией в круглых скобках для строки **SEG1=** в свойствах должен стать активным при подаче на **COM2** логической единицы, а на **SEG1** – логического нуля. Проверяем… (Рис. 111).

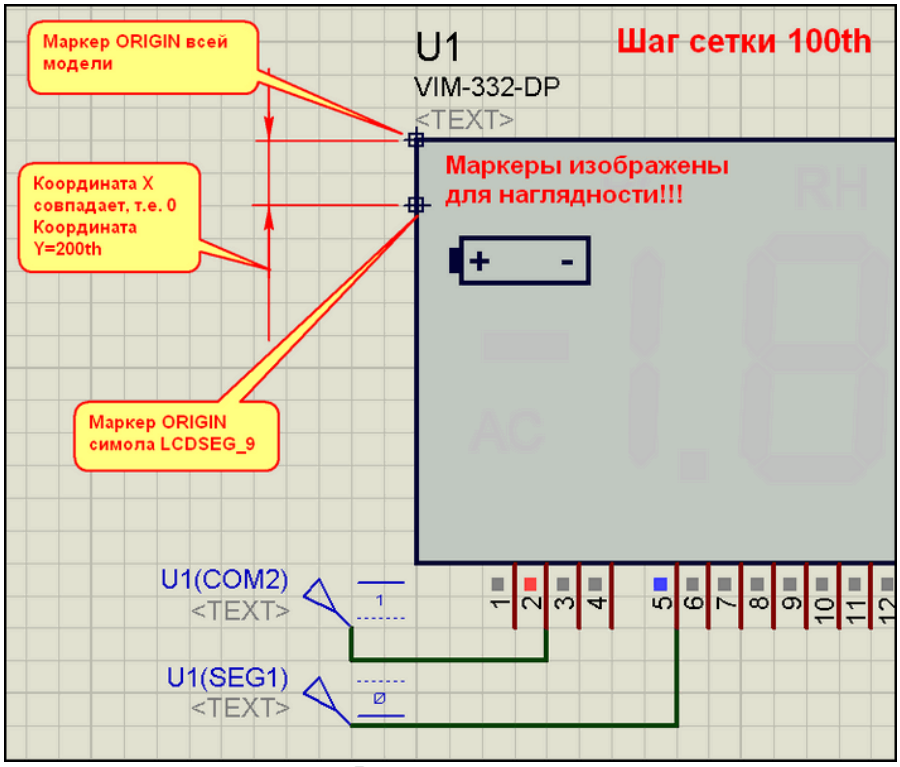


Рис. 111

Как видим, наши предположения полностью подтвердились. Сами изображения маркеров **ORIGIN** я нанес на рисунок для наглядности. По оси X смещение маркера символа отсутствует, по оси Y равно двум клеткам, т.е. при шаге сетки **100th** соответствует числу 200.

Ну и в заключении к этому познавательному материалу, прежде чем мы перейдем к практическому созданию моделей, несколько существенных замечаний.

Первое из них касается описания координат в свойствах модели, а конкретно разделительных запятых. Привычного для обычного текста пробела после запятой, а уж тем более до нее лучше не делать. Это касается как написания координат внутри круглых скобок, так и запятых между секциями. Хотя, если посмотреть на последнюю строчку, мы видим их присутствие в модели в двух местах, но исходя из личного опыта, особенно с более ранними версиями Протеуса, я все же рекомендую их не делать. В ряде случаев наблюдалась неадекватная реакция симулятора на наличие пробелов в свойствах сегментов.

Второе замечание касается относительной координаты по оси Y. Предопределяя желание некоторых любителей покричать: «Вот это глюк! Так глюк!», поясняю – реально в последней картинке смещение по Y должно быть отрицательным – ведь оно направлено вниз, и мы столкнемся с этим фактом при построении своих моделей. Почему же стоит положительное число 200? Программисты меня и автора **LCDMPX** поймут с полуслова – а стоит ли таскать внутри модуля знаковые константы, если они используются только внутри и никак не связаны со всей остальной программой? Поэтому не удивляйтесь, что конструируя свои модели мы будем получать отрицательное смещение по оси Y, но скромно забывать при этом о знаке числа.

Ну и последнее, если Вы заглянете самостоятельно в свойства модели часового индикатора VI-402-DP с установленной галочкой, то увидите, что там для описания свойств строки SEG= имеют только одну секцию с координатами, ведь у него только один общий вывод COM1, зато строк сегментов будет гораздо больше. Желающие могут скачать даташиты на эти индикаторы, они доступны при подключенном Интернете при нажатии кнопки **Data** в свойствах модели, либо через клик правой кнопкой и выборе опции **Display datasheet**. Правда, сразу могу огорчить, на просторах России я этих индикаторов не встречал – не завозят.

К содержанию

8.11. «Фальшивая» точка начала координат - наш помощник в деле создания графики индикаторов. Трансформируем модель VI-402-DP в шестиразрядный индикатор ITS-E0809.

Будем считать, что со структурой модели на основе **LCDMPX.DLL** мы познакомились. Наступил черед практического создания первой самостоятельной модели на основе данной библиотеки. В качестве прототипа для нашей будущей модели я выбрал четырехразрядный часовой ЖК-дисплей **VI-402-DP**, с которым мы познакомились в предыдущем параграфе. А создадим мы на его основе не какую-нибудь «экзотику», а более распространенный на необъятных

просторах нашей Родины шестиразрядный ЖК индикатор ITS-E0809 уже упоминавшийся мною выше китайской фирмы **Intech LCD Group**. Эти индикаторы до сих пор продаются в ряде Интернет-магазинов, как в России – «Платан», «Чип-Дип», так и в ближнем зарубежье – украинский «Космодром».

Но прежде чем мы перейдем непосредственно к созданию модели хочу рассказать об одной «фишке» Протеуса, именно всего Протеуса, т.к. данная опция работает и в ISIS, и в ARES, в ряде случаев значительно облегчающей жизнь при создании графики моделей. А заключается она в преднамеренной установке «фальшивой» точки начала координат на поле проекта. Данная опция описана в HELP по **Universal Keypad Model**, и мы будем ее также использовать при создании моделей клавиатуры, а пока я расскажу – как это работает на практике. Для начала «разберем на запчасти» (**Decompose**) наш прототип **VI-402-DP**. Я рекомендую вам сразу же увеличить параметры листа проекта (**System => Set Sheet Sizes…**), хотя бы до А3, так как создание графики требует простора для творчества. Кроме того, неплохо бы сразу выделить полностью все составляющие разбитой модели и через кнопку **Block Copy** накидать по полю проекта пару-тройку резервных копий разбитой модели, чтобы всегда иметь в запасе «неиспорченные» нашим творчеством запчасти. И еще, перед началом экспериментов убедитесь, что шаг сетки стоит так, как принято по умолчанию – **100th**. Более мелкий шаг при создании моделей обычно необходим только при прорисовке мелких деталей графики, а в данном случае он будет только мешать. Итак, совмещаем курсор мышки с центром маркера **ORIGIN** одной из разбитых моделей в ее левом верхнем углу и нажимаем клавишу с латинской буквой «**O**» (не путать с нулем!!!). Кстати, переключать раскладку клавиатуры в нашем случае совсем не обязательно, Протеус среагирует адекватно и на русскую «**Щ**». Что при этом произойдет, показано на рисунке 112.

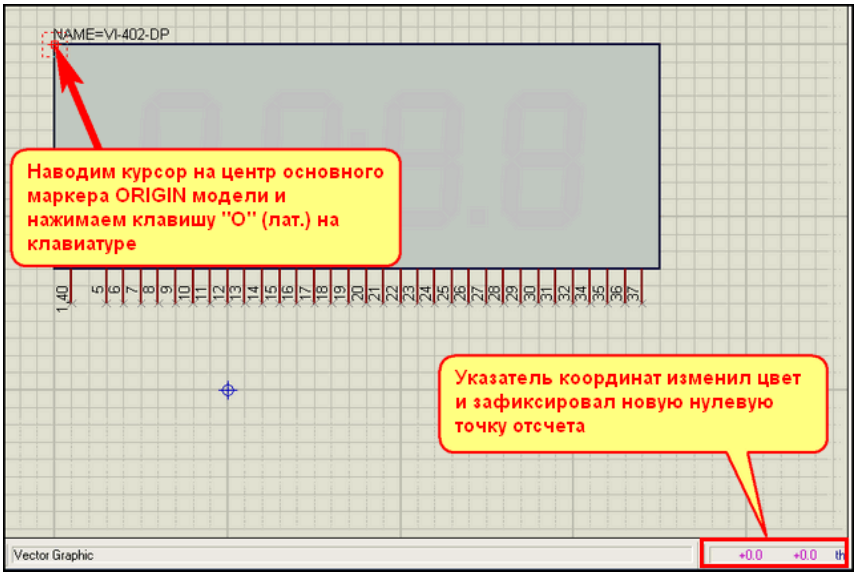


Рис. 112

Как видим, цифровые значения координат в окне программы в правом нижнем углу слегка «полиловели» и приняли нулевое значение. Теперь любое телодвижение мыши будет изменять их значение относительно новой точки отсчета – левого верхнего угла модели или установленного там маркера **ORIGIN**. Если накануне было «слегка принято на грудь» и «рука мыша дрожать устала», то никогда не поздно вернуться к первоначальному отсчету повторным нажатием клавиши «**O**». Затем, собрав волю в кулак и сфокусировав зрение на нужной точке можно повторить операцию. Теперь рассмотрим, зачем нам такие мучения…

Поместим из селектора символов какой-нибудь контрастный (засвеченный) символ на соответствующее ему место первой цифры индикатора. В качестве примера я выбрал символ сегмента «A» (верхнего горизонтального). В селекторе символов он у нас фигурирует, как **LCDSEG_0_1**. Почему я выбрал именно контрастный темный? Да потому, что потом его легко отследить и удалить, хотя при достаточном запасе резервных копий модели можно и не «париться» с этой процедурой. Теперь на этом месте мы его тоже **Decompose**, чтобы увидеть положение маркера самого символа (Рис. 113).

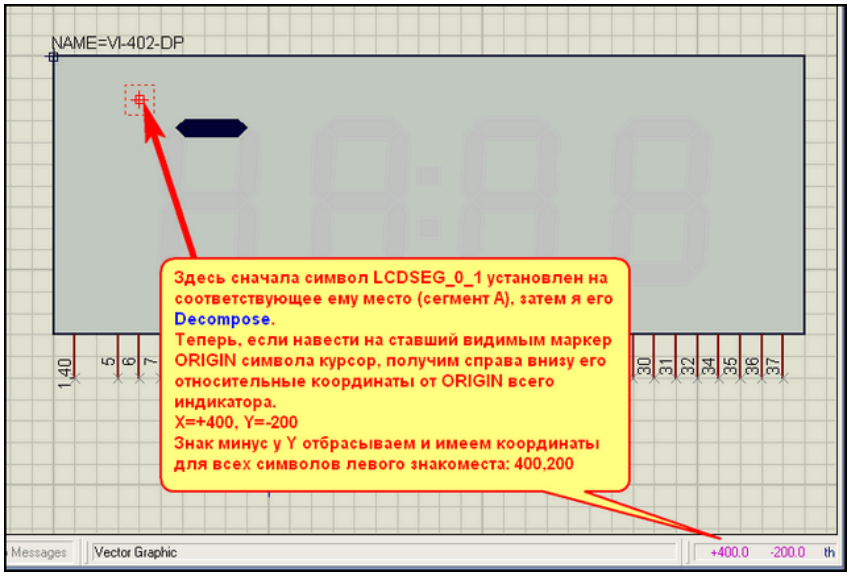


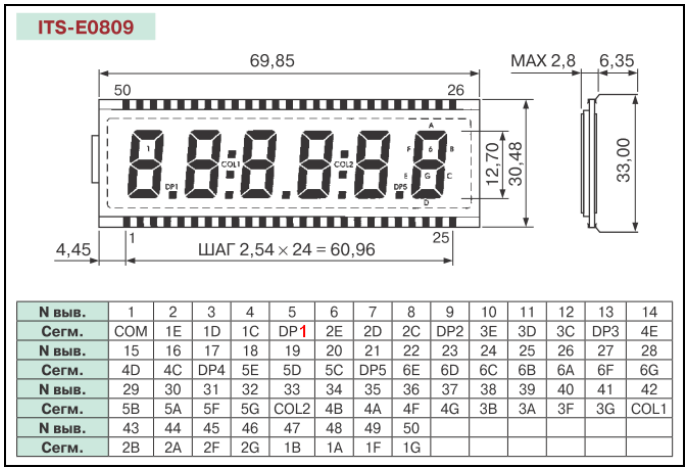
Рис. 113

Наводим курсор на появившийся маркер **ORIGIN** символа и в правом нижнем углу получаем координаты относительного смещения этого маркера от маркера **ORIGIN** всей модели. Как я уже упоминал, поскольку смещение вниз, координата **Y** имеет отрицательное значение, на знак мы не обращаем внимания. Итак, для сегментов первой цифры, мы получили значение **400,200**. Если теперь заглянуть в свойства исходного, не разбитого индикатора **VI-402-DP** при установленном флажке **Edit all properties as text**, то мы увидим, что для ряда сегментов (**SEG1…SEG4** и **SEG29…SEG32**) использована именно эта пара значений координат. Аналогичным образом определяются координаты для символов остальных знакомест (Рис. 114).

Рис. 114

Не надо быть Фурье или Лобачевским, чтобы вывести одну закономерность – для всех символов модели индикатора смещение по оси **Y** составляет **200th**, а интервал по оси **X=700th**. Именно с таким интервалом нам и предстоит расположить дополнительные две цифры нашей будущей модели. Надеюсь, что вы самостоятельно можете проделать эту работу без подробных комментариев. Вкратце, все сводится к следующему: растягиваем вправо тело-подложку на нужную ширину и, выделив нужный участок графики на разобранной модели, копируем его на новое место. После того, как работа проделана, имеет смысл проверить, что мы не промахнулись в графике, а заодно и сверить координаты добавленных элементов, как мы это только что проделали. Соответственно для пятой и шестой цифр они будут **3200,200** и **3900, 200**. Если у вас получилось по другому, значит где-то вы сдвинули знакоместо.

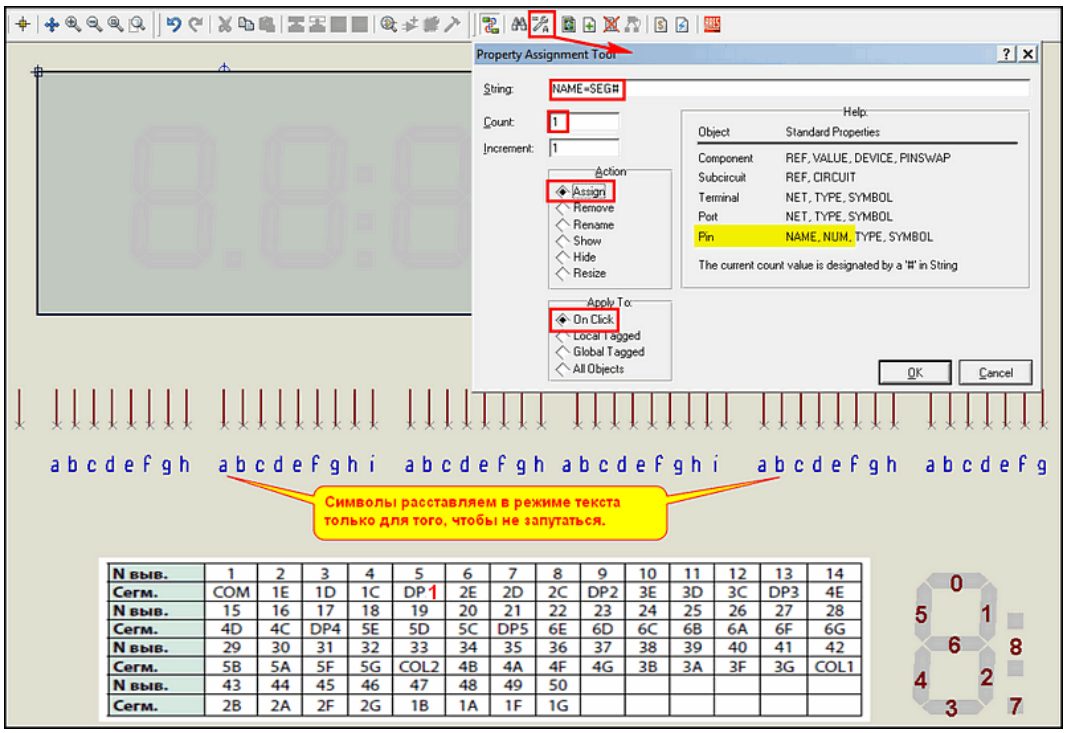
Ну, вот мы и разобрались с «крыльями» нашей модели, пора приниматься за «ноги и хвосты». Для этого нам потребуется даташит индикатора ITS-E0809, который доступен на сайте фирмы **Intech LCD Group**, по ссылке в предыдущем параграфе. Я же воспользуюсь данными на индикатор из каталога с сайта www.platan.ru. В нем они расположены в более компактной форме и помещаются здесь на одном скриншоте из данной документации (Рис. 115). Меня в данном случае интересует нижняя таблица соответствия выводов индикатора сегментам. Но, сразу предупреждаю, там есть ошибка, которую я поправил красным цветом.



N выв.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Сегм.	COM	1E	1D	1C	DP1	2E	2D	2C	DP2	3E	3D	3C	DP3	4E
N выв.	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Сегм.	4D	4C	DP4	5E	5D	5C	DP5	6E	6D	6C	6B	6A	6F	6G
N выв.	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Сегм.	5B	5A	5F	5G	COL2	4B	4A	4F	4G	3B	3A	3F	3G	COL1
N выв.	43	44	45	46	47	48	49	50						
Сегм.	2B	2A	2F	2G	1B	1A	1F	1G						

Рис. 115

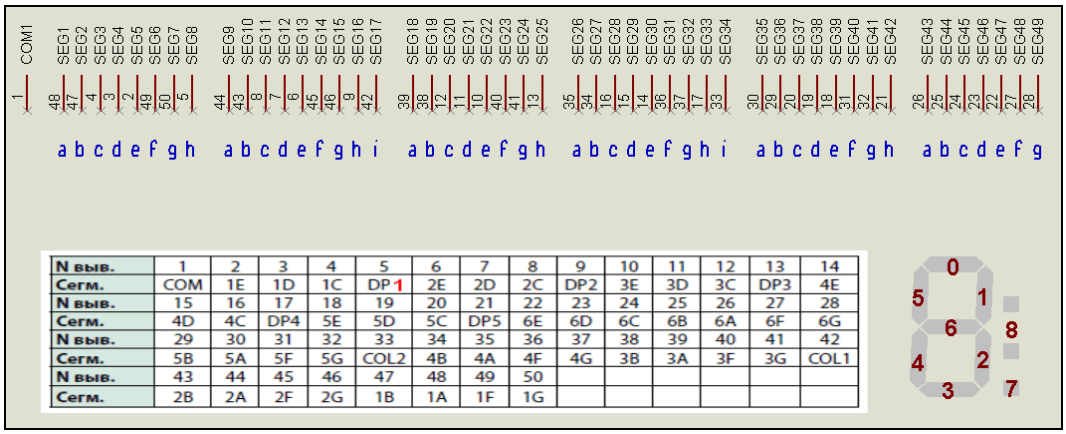
Далее можно идти различными путями. Кому то может понравиться вариант расположения выводов так, как у реального индикатора, а кому то захочется расположить их в один ряд. В данном случае это возможно только с шагом **50th**. При этом придется скрыть имена и номера выводов, а для удобства расположить их группами, соответствующими каждой цифре. Итак, приступаем к расстановке выводов. Хотя я и таскал из картинки в картинку выводы от прототипа, но сейчас рекомендую их удалить. Дело в том, что при большом количестве «ног» удобнее и быстрее воспользоваться опцией **Property Assignment Tools** (**PAT**), но у нее иногда бывают мелкие глюки именно при замене одного имени на другое. Чтобы обезопасить себя от этого проще поставить свежие выводы без номеров и имен, с ними проблем не бывает, тем более времени это отнимет немного. Поставили пяток выводов, потом обвели их, выделили и … **Block Copy** столько сколько нужно. Сначала я расположу выводы группами с шагом **100th** и на некотором расстоянии от модели, а когда закончу нумерацию и наименование, пододвину на нужные места. Еще один прием, которым я часто пользуюсь в таких случаях, это сохраняю в формате **BMP** картинку с расположением выводов в данном случае нижнюю таблицу с рисунка 115 и втаскиваю ее в проект с нужным масштабом через **File => Import Bitmap**. Этот прием исключит из творческого процесса постоянное перепрыгивание из окна в окно для проверки соответствия номера, названию. Можно, конечно, распечатать на бумаге, тогда будете прыгать глазами экран-бумага-экран. Наверное, этот вариант имеет право на жизнь – зарядка для глаз сохранит зрение, но макулатуры в хозяйстве добавится. Там же имеет смысл расположить одно графическое изображение всех сегментов с надписанными номерами, и просто текстовые строки под группами выводов, которые дополнительно облегчат жизнь при расстановке номеров и имен выводов (Рис.116).



N выв.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Сегм.	COM	1E	1D	1C	DP1	2E	2D	2C	DP2	3E	3D	3C	DP3	4E
N выв.	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Сегм.	4D	4C	DP4	5E	5D	5C	DP5	6E	6D	6C	6B	6A	6F	6G
N выв.	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Сегм.	5B	5A	5F	5G	COL2	4B	4A	4F	4G	3B	3A	3F	3G	COL1
N выв.	43	44	45	46	47	48	49	50						
Сегм.	2B	2A	2F	2G	1B	1A	1F	1G						

Рис. 116

Итак, запускаем **PAT**, в строке **String** набираем **Name=SEG#** а в строке **Count** меняем стартовое значение с **0** на **1**, как показано на том же рисунке 116. Как вы уже догадались, на данном этапе мы будем использовать **PAT** для операций с выводами (**Pin**), конкретно сначала для имен **NAME**, затем для номеров **NUM**, я выделил желтым цветом это в **Help** окна **PAT**. Убеждаемся, что **Action** оставлено по умолчанию **Assign** (назначить) и **Apply To** – **On Click** (по клику мыши), нажимаем кнопку **OK**. Теперь пробегаемся кликами мышкой по всем выводам сегментов слева направо, общий вывод **COM1** мы назовем вручную позже. Затем снова вызываем **PAT** и набираем в строке **String** значение **NUM=#**, опять исправляем **Count** с **0** на **1** и опять **OK**. Дальше строго руководствуемся таблицей и расставляем номера выводов в соответствии с ней, не забывая присвоить номер **1** отдельно стоящему выводу **COM1**. Что после этого получилось, показано на рисунке 117.



N выв.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Сегм.	COM	1E	1D	1C	DP1	2E	2D	2C	DP2	3E	3D	3C	DP3	4E
N выв.	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Сегм.	4D	4C	DP4	5E	5D	5C	DP5	6E	6D	6C	6B	6A	6F	6G
N выв.	29	30	31	32	33	34	35	36	37	38	39	40	41	42
Сегм.	5B	5A	5F	5G	COL2	4B	4A	4F	4G	3B	3A	3F	3G	COL1
N выв.	43	44	45	46	47	48	49	50						
Сегм.	2B	2A	2F	2G	1B	1A	1F	1G						

Рис. 117

Теперь тщательно проверим, что мы не наделали ошибок и воспользуемся все тем же **PAT**, чтобы скрыть имена и номера выводов, потому что когда они будут расположены с шагом **50th**, текст соседних выводов будет перекрываться. Для этого выделяем мышью все выводы модели и вновь давим кнопку **PAT** в верхнем меню. В строке **String** набираем просто **NAME**, устанавливаем переключатель **Assign** в положение **Hide** (скрыть), а **Apply To** в положение **Local Target** и нажимаем **OK** (Рис. 118). Аналогично повторно выделяем выводы и повторяем вызов **PAT**, но вместо **NAME** ставим **NUM**, чтобы скрыть номера выводов. Вы наверное уже догадались, что если вместо **Hide** выбрать **Show** (показать), то вместо скрытых имен/номеров, получим видимые. Самое время освоить **Property Assignment Tools** тем, кто этого еще не сделал.

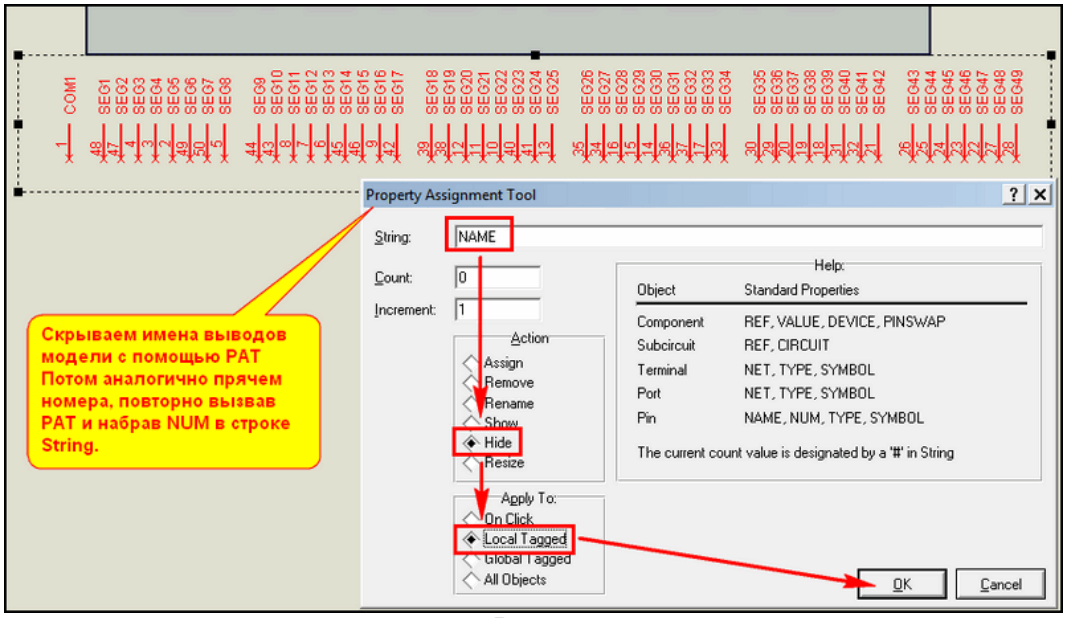


Рис. 118

Итак, я установил шаг сетки **50th**, сдвинул выводы в группах и придвинул их к модели (Рис 119). Попутно я добавил на изображение модели текстовые строки с подсказкой о назначении выводов. Теперь графика полностью готова к созданию модели индикатора **ITS-E0809**.

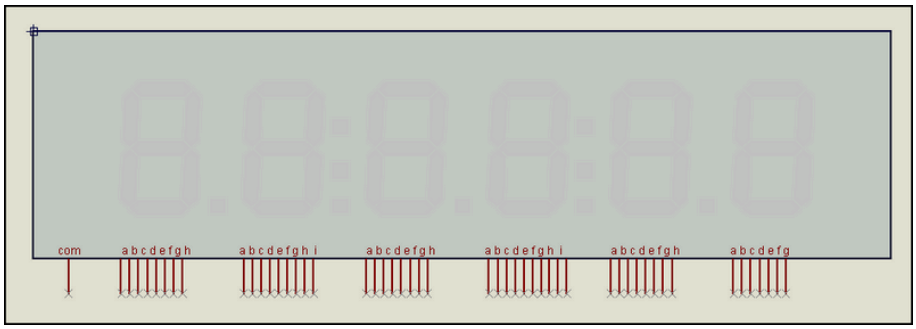


Рис. 119

Настал черед процедуры **Make Device**. Причем на первом этапе я не стану добавлять координаты сегментов, а просто создам устройство с базовым набором свойств для **LCDMPX.DLL**. Мы уже столько раз проходили **Make Device** в процессе изучения Протеуса, что, надеюсь, нет смысла приводить скриншоты с подробным видом задаваемых свойств, достаточно описать – что и где задается. На первой вкладке **Make Device** задаем нашему девайсу имя – **ITS-E0809**, префикс – **U** (а кто желает может и **H** или **HG**) , а также задаем параметры активного компонента: **Symbol Name Stem** – **LCDSEG**, количество – **9**, устанавливаем флажки бит-зависимости и связи с DLL. Окно с корпусом оставляем не заполненным, а в третьем окне добавляем следующие свойства:

Name	Description	Type	Type	Default Value	Visibility
PRIMITIVE	Primitive Type	String	Hidden	DIGITAL,LCDMPX	Hide Name & Value
MODDLL	VSM Model DLL	String	Read Only	LCDMPX.DLL	Hide Name & Value
TTRIGMIN	Trigger Time	String	Hidden	1ms	Hide Name & Value
TRACE	Diagnostic Messages	Trace mode	Hidden	Warnings Only	Hide Name & Value

Последнее, относящееся к диагностике, можно и не добавлять, чем я и воспользовался в прилагаемом примере. В последнем окне выбираем категорию **Optoelectronics**, подкатегорию **LCD Panels Dysplays**, добавляем через кнопку **New** производителя **Intech LCD Group**, и что-нибудь типа **6 Digits LCD Panel** в описание девайса. Сохраняем все это в нужной библиотеке. Как всегда, свободной для записи является **USRDVC**, но при желании можно добавить модель к существующим дисплеям – там еще 13 позиций свободно, или создать свою библиотеку дисплеев. О том, как снять защиту записи с библиотек или создать свою говорилось ранее.

В принципе, модель уже «работоспособна» с точки зрения того, что при симуляции не выскакивают ошибки, но и индикации соответственно никакой. Почему я не стал сразу добавлять координаты сегментов? Ну, во-первых, потому что это можно осуществить двумя способами, и я их сейчас опишу, а во-вторых, потому что их много и лучше добавлять их не скопом, а в несколько приемов, проверяя работоспособность на каждом этапе.

Итак, вариант первый – «классический». Описание сегментов добавляем через третье окно **Make Device**, при этом нам придется каждый раз вводить вручную все параметры. Процесс представлен на рисунке 120.

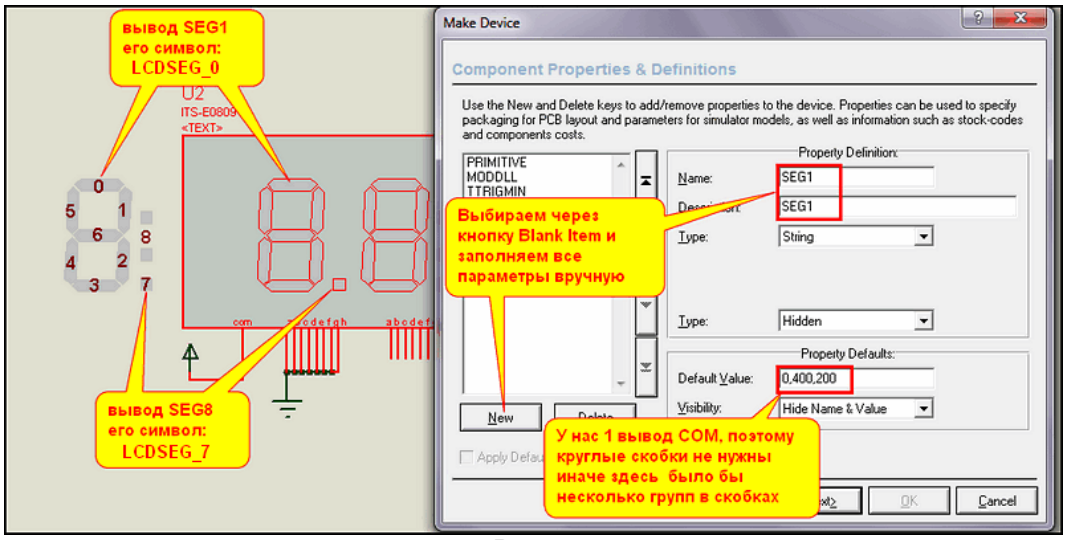


Рис. 120

Для созданной нами модели **ITS-E0809** запускаем **Make Device**, переходим в третье окно и через кнопку **New** начинаем добавлять **Blank Item**. Для каждого сегмента нам предстоит руками задать **Name**, **Description**, установить **Type=Hidden** и в графе **Default Value** задать номер символа и его координаты вручную. Для всех сегментов, составляющих первую цифру координаты одинаковы и равны **400,200**, а вот номер символа будет отличаться. Для верхнего сегмента «a» первой цифры – **SEG1** он будет равен **_0**, а десятичной точке – **SEG8** соответствует символ **LCDSEG_7**. Данный метод хорош тем, что в любой момент можно прекратить добавление сегментов, пройти процедуру **Make Device** до конца, сохранить изменения и проверить – что и как работает. Для примера на рисунке 121 приведен момент проверки, когда в модели добавлены параметры только для первых четырех сегментов **SEG1…SEG4**, т.е. сегменты «**a, b, c, d**» первой цифры. Хотя активные сигналы поданы на все выводы первой цифры, работают только эти сегменты. Таким образом, можно в любой момент проверить свои действия. Но, как видим, процедура ручного ввода при большом количестве активных элементов – довольно рутинное и утомительное занятие.

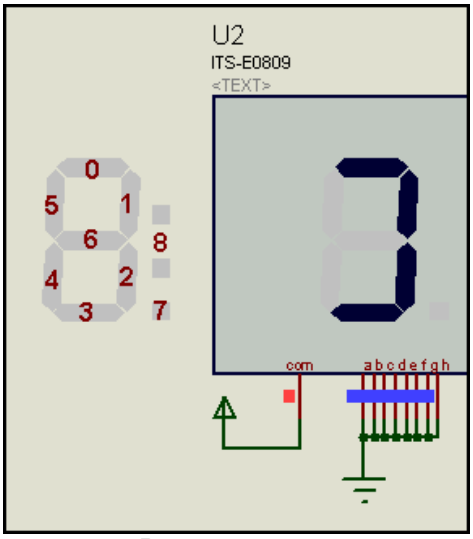


Рис. 121

Второй вариант заключается в том, что данные по сегментам заранее готовятся в любом текстовом редакторе и добавляются сразу, непосредственно в окне **All Properties** свойств нашей модели (Рис. 122). Тут тоже возможен самоконтроль на любом этапе, а процедура **Make Device** запускается только один раз, когда все сегменты добавлены и проверены.

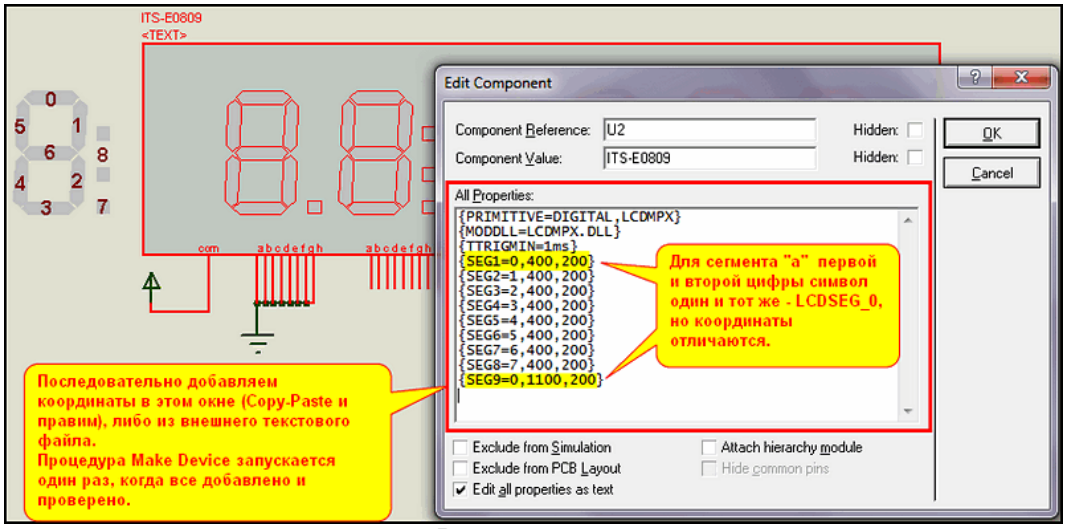


Рис. 122

Вспомним, что для того чтобы параметры были скрытыми (**Hidden**) достаточно заключить их в фигурные скобки. Но это относится только к отображению параметров в окне проекта на поле чертежа на месте серого **<TEXT>**. За отображение же параметров в окне свойств модели отвечает второй сверху параметр **Type** на третьей вкладке **Make Device**. При таком способе добавления свойств он по умолчанию остается **Normal**. Это означает, что если мы не изменим его на третьей вкладке для каждого сегмента, выбрав из раскрывающегося списка **Hidden**, то в конечном итоге

получим в готовом девайсе параметры **SEG** выделенными в отдельные строки. На рисунке 123 приведен пример как это выглядит при вызове **Edit Device Properties** для первых четырех **SEG**.

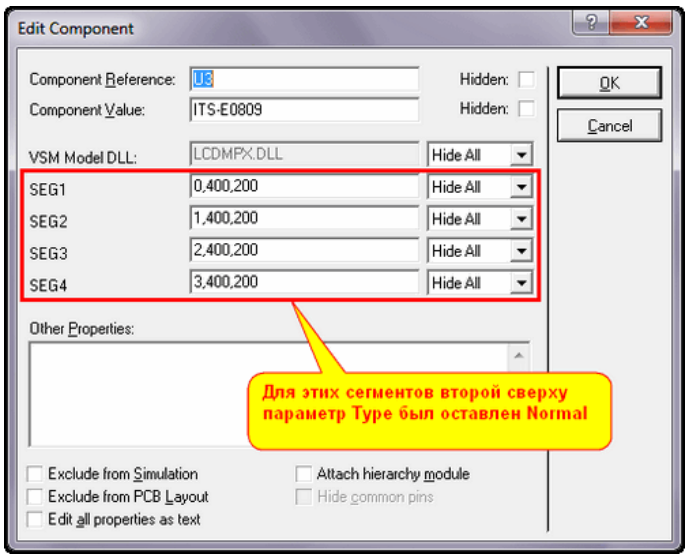


Рис. 123

А представьте, что таких строк будет почти полста? Окошко растянется больше чем экран компьютера. Поэтому, при этом варианте при запуске **Make Device** необходимо на третьей вкладке пробежать мышкой по всем добавленным **SEG** и изменить второй сверху **Type** на **Hidden**. Все же это быстрее, чем набирать на клавиатуре.

Можно пойти и на маленькую военную хитрость, чтобы обмануть Протеус. Заглянем в текстовый скрипт «разобранного» индикатора **VI-402-DP**. Там вначале мы можем найти строки вида:

{SEG1="SEG1",HIDDEN STRING}
{SEG2="SEG2",HIDDEN STRING}

Находятся эти строки в разделе, начинающемся со строки **{*PROPDEFS}**, а сами свойства в разделе, начинающемся со строки **{*COMPONENT}**. Таким образом, можно заранее подготовить аналогичный текст в текстовом редакторе для наших 49 сегментов и добавить в окно **All Properties** перед процедурой **Make Device**. Применительно к первым четырем сегментам рисунка 123 это будет выглядеть так, как показано на рисунке 124.

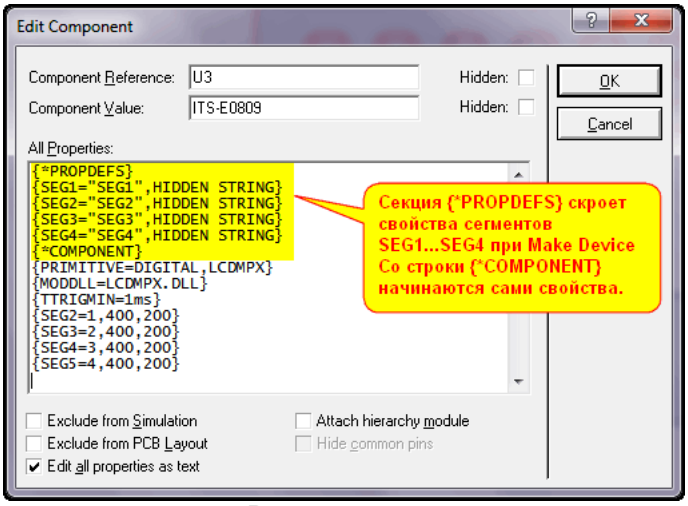


Рис. 124

Конечно же, второй способ удобнее и практичнее, и лично я всегда пользуюсь им. Если данные готовятся в хорошем текстовом редакторе, то всегда можно воспользоваться «продвинутыми» функциями автозамены в выделенном тексте и прочими удобствами редактирования, а не заниматься тупым «копи-пастом» с последующей рутинной ручной правкой.

Вот, вроде, и все хитрости создания собственной модели ЖК индикатора на основе **LCDMPX.DLL**. Как вы уже поняли, главным ее отличием от **LEDMPX** является возможность размещения сегментов по двум координатам, что дает больше возможностей для создания своих

моделей. Кроме того, в моделях на основе **LEDMPX** выводы **SEG** для каждого «знакоместа» с общим выводом **COM** индивидуальны, а не объединены, как в **LEDMPX**. Ну и еще хочу заметить, что в рассматриваемом примере мы воспользовались стандартной расцветкой символов для ЖК (черно-серым вариантом), но это совсем не догма, можно использовать символы и подложку с другой цветовой гаммой и на основе **LCDMPX** сделать модели светодиодного типа. Единственное, что пока тормозит – это ограничение на 64 сегмента, которое, надеюсь, когда-нибудь Лабцентер расширит, как это было со светодиодной библиотекой.

Ну и в заключение несколько слов о том как добавить даташит к модели. Для этого на соответствующей вкладке процедуры **Make Device** указываем имя файла даташита. В моем случае это **E0809.pdf** (Рис. 125).

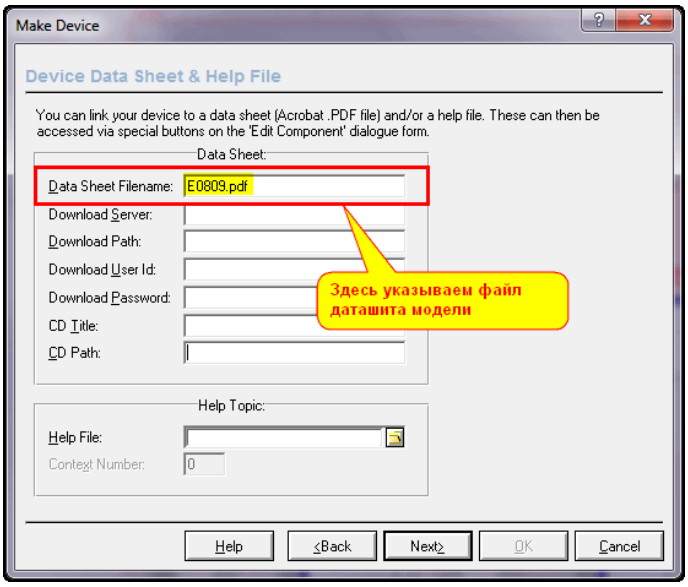


Рис. 125

Это совсем не лишняя информация, особенно, когда у модели скрыты номера и имена выводов. Сам файл даташита должен помещаться в определенной директории. Как определить в какой именно – ясно из рисунка 126. Там показан путь по умолчанию для **Windows 7**, соответственно у «счастливых» обладателей **Windows XP** он будет отличаться. Этот путь в любой момент можно изменить и тогда все скачанные из Интернета даташиты Протеус будет помещать в другую, указанную вами папку. Только не забудьте туда переместить файлы, которые Вы скачали ранее. Ну и конечно в нее же помещаем файл даташита нашей модели, который я поместил во вложении. Если этого не сделать, то Протеус будет пытаться скачать его с сайта **Labcenter**, где его нет и в помине.

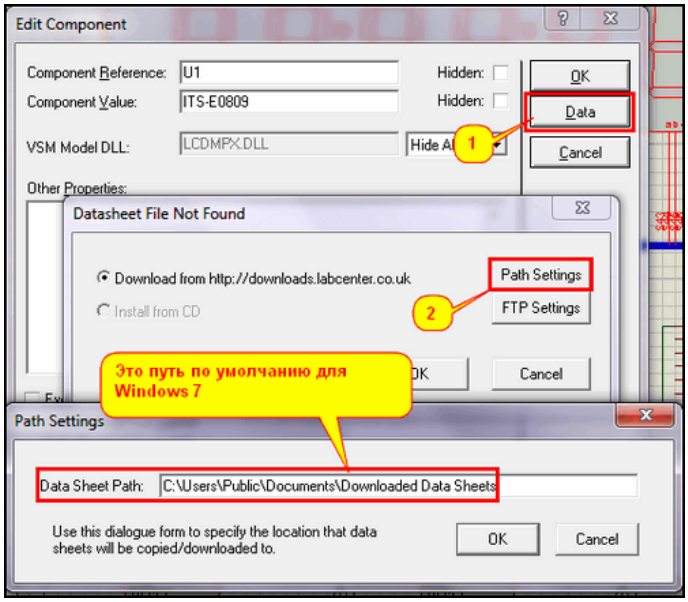


Рис. 126

Во вложении отдельно представлен проект с последовательностью создания графической модели и тестовый проект с пробной симуляцией готового девайса. Конечно же, никаких готовых файлов модели там нет. Если вам необходима модель **ITS-E0809** в ваших установленных библиотеках, достаточно тупо пройти всю процедуру **Make Device** от начала до конца для готовой модели из проекта **TEST.DSN**, ничего не меняя. Ну, можете добавить файл даташита на соответствующей вкладке, и в конце выбрать отличную от **USRDVC** библиотеку, если она у вас уже существует и не защищена от записи.

Мы же далее рассмотрим некий «симбиоз» ЖК модели на основе **LCDMPX** и схематичной модели контроллера для восполнения отсутствующих в библиотеках реальных индикаторов, например COG с контроллером **ML 1001**.

К содержанию

8.12. Реализация «составной» модели ЖК индикатора TIC5231 на основе схематичной модели COG драйвера ML1001 и модели индикатора на основе LCDMPX.DLL в Протеусе.

Возможность воспроизвести в Протеусе обычный сегментный ЖКИ – это конечно замечательно, если количество сегментов невелико, а как быть если мы имеем ЖКИ на 6 или 8 цифр. Занимать под управление таким ЖКИ несколько портов микроконтроллера – это расточительство. В то же время уже давно промышленно выпускаются драйверы для управления ЖКИ как в виде отдельных микросхем, так и интегрированные на стеклянную подложку самого индикатора. Именно последние и носят название – **COG** от английского **Chip On Glass** (микросхема на стекле). Характерным представителем таких драйверов является **ML1001**. На ее основе в свое время тайваньской фирмой **Ampire** была выпущена серия цифровых сегментных ЖК индикаторов TIC, которые еще до сих пор встречаются в продаже. Реализацию одного из них 6,5-разрядного **TIC5231** мы рассмотрим ниже.

Эта модель была разработана около года назад по просьбе одного из участников форума. К сожалению, модели на основе **LCDMPX** поддерживают управление только от пинов **COM** и **SEG**, расположенных на основном листе проекта, и попытка присоединить к модели ЖКИ контроллер, схема которого реализована на дочернем листе не удалась. Именно поэтому родился такой симбиоз: «мухи отдельно – котлеты отдельно». Отдельно была сформирована модель ЖКИ со стандартными ножками **COM** и **SEG** и отдельно сделана схематичная модель COG контроллера **ML1001**.

Сама модель индикатора (Рис. 127) никаких особенностей не имеет. Стандартный ЖКИ, имеющий один вывод **COM** и 40 выводов **SEG**.

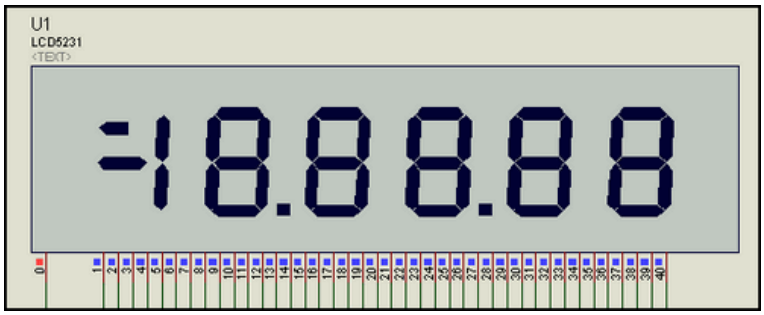


Рис. 127

Единственное, над чем пришлось потрудиться, это создать графические символы для самого левого «половинчатого» разряда (Рис. 128). В результате появились символы отдельно стоящего минуса, урезанного символа надчеркивания и полный знак единицы в виде одного символа, которые и составляют самое левое знакоместо индикатора **TIC5231**.

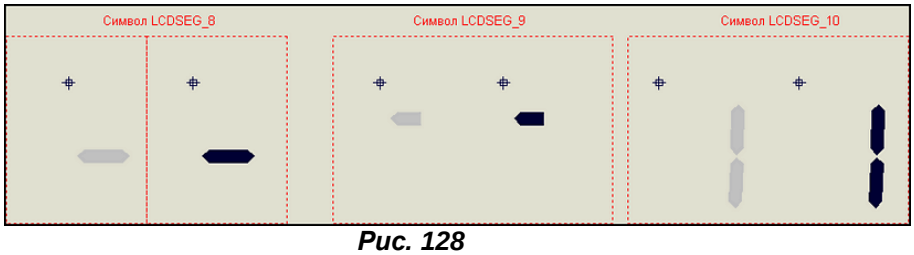


Рис. 128

Первоначальный вариант графики имел полностью 41 вывод, но немного поразмыслив, я решил, что поскольку отдельно такой индикатор практической ценности не представляет, есть

смысл оформить выводы сегментов шиной на 40 разрядов. Это позволяет более компактно располагать индикатор и присоединенный к нему контроллер в проекте.

Хочу также обратить внимание на несколько нестандартную нумерацию сегментов в данном индикаторе (Рис. 129).

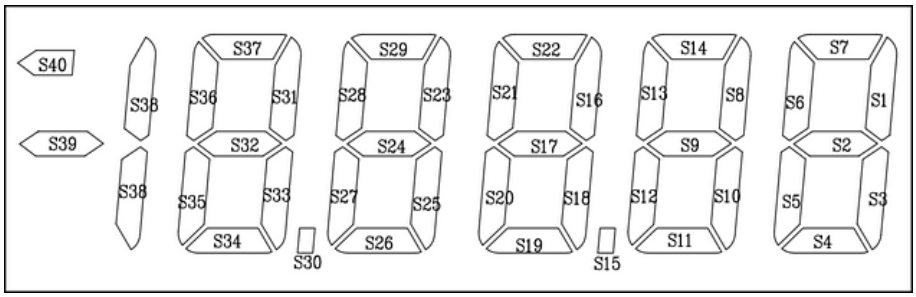


Рис. 129

Полный даташит находится во вложении, а из расположения сегментов на рисунке из него мы видим, что номер 1 принадлежит сегменту «**b**» самого правого разряда индикатора, номер 2 – сегменту «**g**», хотя для нас логичнее было бы, чтобы это были сегменты «**a**» и «**b**» соответственно. Об этом не стоит забывать, применяя реальный индикатор, иначе вместо цифр получите полную чехарду с сегментами. В остальном графическая модель особенностей не имеет, т.к. выводы нужны только для «внутреннего» соединения с контроллером, я дал им наименование **SEG** и нумерацию в соответствии с номерами сегментов реального индикатора. В окончательном варианте это просто 40-разрядная шина **SEG[1..40]**.

Теперь рассмотрим сам драйвер **ML1001** и реализацию модели для него. Во вложении имеется как оригинальный даташит от компании **Minilogic Device Corporation**, так и его перевод выполненный Игорем Данко. Сайт автора перевода на данный момент не работает, поэтому я принял решение включить всю документацию во вложение.

Блок-схема драйвера приведена на рисунке 130.

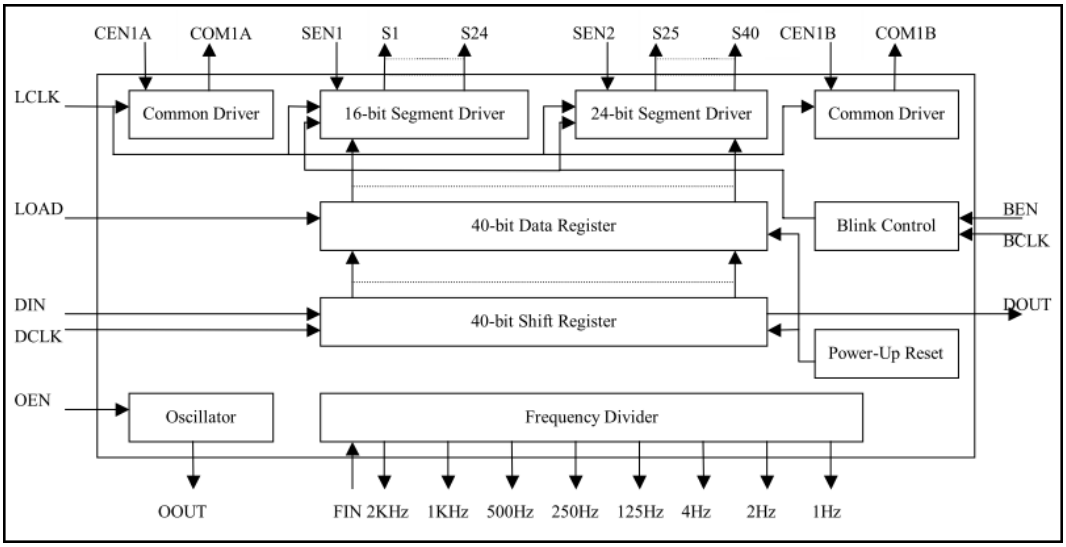


Рис. 130

Драйвер имеет в своем составе 40-разрядный сдвиговый регистр с последовательным вводом данных. Данные загружаются с входа **DIN** по тактовой частоте на входе **DCLK**. По сигналу на входе **LOAD** данные переносятся в 40-битный регистр-защелку, с выхода которого распределяются по сегментам индикатора с помощью дополнительных встроенных управляющих схем. С помощью выхода **DOUT** драйвера можно соединять последовательно, что дает возможность наращивать разрядность индикатора с кратностью 40. Конкретно для индикатора **TIC5231** используется один чип, но учитывая, что модель **ML1001** может использоваться и для индикаторов с большим количеством разрядов, есть смысл реализовать наличие этого вывода в модели. Кроме того, в реальном чипе имеются встроенный генератор и делитель частоты, но с точки зрения моделирования они нас мало интересуют. Если рассмотреть реальную реализацию в индикаторе (Рис. 131), то имеются только 7 внешних выводов, из которых мы еще не рассмотрели 3.

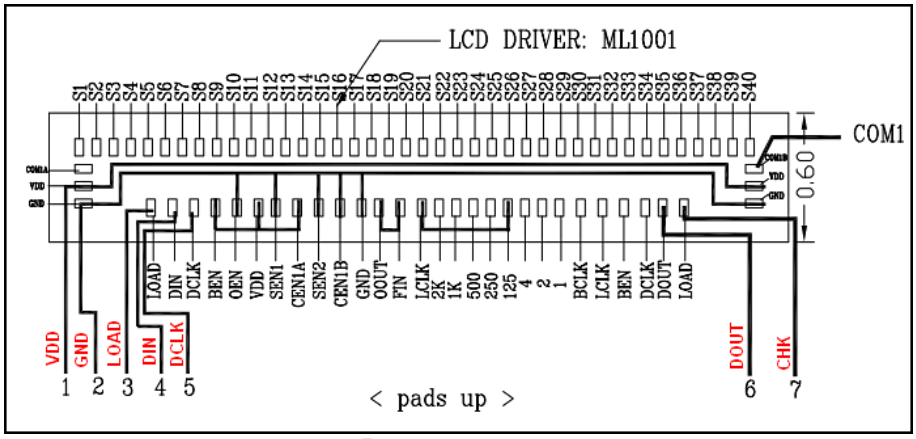


Рис. 131

Ну, с выводами **VDD** и **GND** все ясно и без комментариев – это выводы питания, а вывод **7**, обозначенный в даташите индикатора, как **CHK** – это фактически дубль вывода **LOAD**. Таким образом, для реализации модели нам нужны всего четыре «видимых» вывода: **DIN**, **DCLK**, **LOAD** и **DOUT**, ну и, конечно, выходная шина на 40 сегментов. В итоге, для начального тестового варианта графическая модель **ML1001** имеет вид, представленный на рисунке 132.

Рис. 132

Для тестирования лучше использовать вариант с выводами, а после убрать выводы выходов и заменить их шиной. Теперь перейдем к внутренней реализации модели **ML1001**, т.е. к тому «фаршу», из которого мы слепим нашу «котлету» на дочернем листе. Здесь тоже особых премудростей нет, на дочернем листе я разместил 4 стандартных примитива сдвиговых десятиразрядных регистров и 5 восьмиразрядных регистров-защелок. Фрагмент схемы с дочернего листа приведен на рисунке 133.

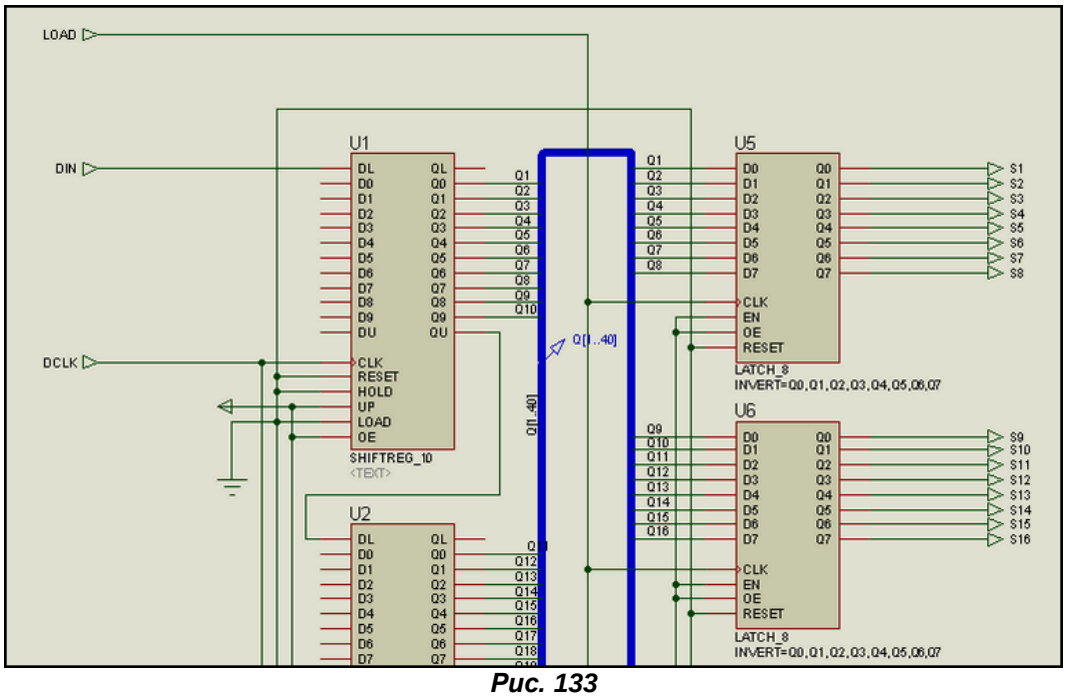


Рис. 133

Для того, чтобы согласовать по уровням сигналы с моделью ЖКИ выходы регистров-защелок инвертированы. С выхода **QU** четвертого регистра сдвига сигнал подается на ножку **DOUT**. После тестирования этого варианта с дочернего листа был сформирован файл **ML1001.MDF**. Он

будет одинаковым как для графической модели с отдельными выводами на выходе, так и с 40-разрядной шиной.

Для того, чтобы использовать данные модели в своих проектах, поместите файл **ML1001.MDF** из любой папки вложения в папку **MODELS** установленного Протеуса, а для моделей индикатора **TIC5231** и драйвера **ML1001** из проекта **TEST.DSN** в папке **ML1001_final_bus** проведите **Make Device** от начала до конца с сохранением в нужной библиотеке. Мы же в следующем параграфе рассмотрим как «обмануть» Протеус, смоделировав ЖКИ с числом сегментов свыше 64.

К содержанию

8.13. Модель ЖК индикатора TIC8148 (TIC55) на основе схематичной модели двойного драйвера ML1001 со встроенным генератором.

В качестве «подопытного» кролика для реализации модели с количеством сегментов более 64 я выбрал тоже достаточно распространенный индикатор **TIC8148**. У него есть брат-близнец – **TIC55**, так что модель можно смело использовать и для того и для другого индикатора. Данный индикатор представляет из себя 8 восьмерок с точками и спецсимволами в виде 8 галок в нижней строке. Таким образом, мы имеем в сумме 72 светящихся сегмента, что явно превышает возможности **LCDMPX.DLL** для одного общего входа **COM**. Даташит, как обычно, находится во вложении, а на рисунке 134 из этого даташита я выделил цветом значимые для нас места. Итак, нумерация цифр самого индикатора справа-налево, т.е. самая левая - цифра 8. Буквенное обозначение сегментов совпадает с общепринятым, но вот десятичная точка обозначена как **P** (по-видимому, от Point), а символом **H** обозначен дополнительный символ - галочка в нижней строке.

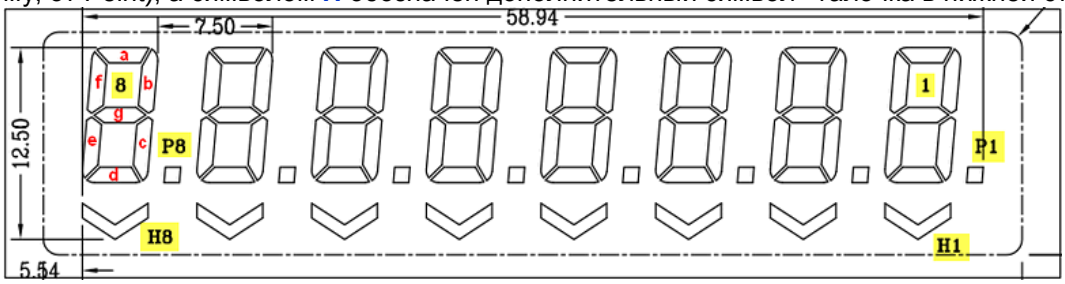


Рис. 134

Вы, конечно, догадались, что 72 сегмента превышают не только возможности **LCDMPX**, но и возможности одного драйвера **ML1001** (40 сегментов), поэтому в реальном индикаторе их 2, включенных последовательно, как рассматривалось выше. А вот если глянуть на таблицу распределения сегментов из того же даташита (Рис. 135), то тут опять начинается «китайская грамота». Немудрено, ведь на Тайване те же лица, только в профиль. Итак, согласно таблице, первому выходу **S1** первого драйвера соответствует сегмент **8D** – нижний горизонтальный самой левой цифры.

PIN	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17	S18	S19	S20
COM1	8D	8E	8G	8F	8A	8B	8C	H8	P8	7D	7E	7G	7F	7A	7B	7C	H7	P7	6D	6E
PIN	S21	S22	S23	S24	S25	S26	S27	S28	S29	S30	S31	S32	S33	S34	S35	S36	S37	S38	S39	S40
COM1	6G	6F	6A	6B	6C	H6	P6	5D	5E	5G	5F	5A	5B	5C	H5	P5	4D	4E	4G	4F
PIN	S41	S42	S43	S44	S45	S46	S47	S48	S49	S50	S51	S52	S53	S54	S55	S56	S57	S58	S59	S60
COM1	4A	4B	4C	H4	P4	3D	3E	3G	3F	3A	3B	3C	H3	P3	2D	2E	2G	2F	2A	2B
PIN	S61	S62	S63	S64	S65	S66	S67	S68	S69	S70	S71	S72	S73	S74	S75	S76	S77	S78	S79	S80
COM1	2C	H2	P2	1D	1E	1G	1F	1A	1B	1C	H1	P1								

Рис. 135

Соответственно по первому тактовому импульсу на входе **DCLK** мы заносим данные для него, далее согласно строкам таблицы, вплоть до 72-го тактового импульса, который соответствует десятичной точке самой правой цифры, т.е. №1. Об этом следует помнить при написании программ для управления данными индикаторами. Еще хочу обратить ваше внимание на то, что дополнительный символ галочки в этой последовательности предшествует символу десятичной точки данного знакоместа, т.е. перебираются все сегменты, затем галка и уже последней – десятичная точка. Это тоже накладывает некоторые особенности на написание программ. В частности, если даже мы не используем в своем девайсе символы нижних галок. Допустим, для хранения информации о знакоместе используется байтовая переменная: семь бит – сегменты и

один бит – информация о точке. При последовательном выводе на индикатор нам придется перед выводом информации о десятичной точке вставлять «пустышку» для сегмента нижней галочки данного знакоместа. Так, что-то я отклонился от темы, это уже особенности программирования, а мы вернемся к нашей модели и начнем с графики.

В качестве символов сегментов я использовал все те же символы от «разобранного» **VI-402-DP**, но пришлось добавить символ нижней галочки. Кроме того, у нас в индикаторе теперь 8 цифр и для двух правых знакомест появились новые координаты – **4600,200** и **5300,200** (Рис. 136). Шаг по оси **X** по-прежнему **700**. Ну и еще небольшой «финт» с выводами. Сегменты я сразу же загнал в шину на 24 разряда **S[1..24]**, а общих выводов сделал три **COM1**, **COM2** и **COM3**, т.е. мы получим индикатор на 72 сегмента, но выводами **COM** надо управлять попеременно, а сегменты мультиплексировать. Собственно, для этого и существует **LCDMPX.DLL**.

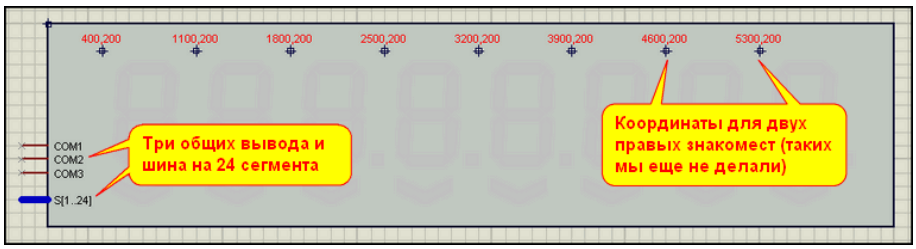


Рис. 136

Теперь небольшое лирическое отступление на тему – почему именно 3 вывода **COM**, ведь для 72 сегментов по логике достаточно было бы и двух. Я тоже первоначально надеялся, что хватит двух, но в ходе разработки модели выявился один неприятный момент **LCDMPX.DLL**. Если количество сегментов для одного вывода **COM** превышает 32, а этих выводов два или более, то при мультиплексировании начинает наблюдаться моргание всех сегментов, относящихся к выводам **COM2**, **COM3** и т.д. И избавиться от этого эффекта всеми известными мне приемами так и не удалось. Так что на будущее примите совет, при моделировании ЖК индикаторов с одним общим выводом можно использовать до 64 сегментов, при моделировании с двумя и более общими выводами количество сегментов для одного общего вывода не должно превышать 32. Чтобы не быть голословным, я оставил во вложении первоначальный вариант с двумя **COM** в папке **BAD_Model_Test**. Просто запустите симуляцию и вы увидите, что правая часть сегментов индикатора (с 41-го по 72-й) вместо постоянного свечения мерцает. Поэтому в окончательном варианте я поставил 3 вывода **COM**, при этом количество сегментов для одного общего вывода получилось 72/3=24, отсюда и соответствующая шина **S[1..24].**

Теперь нам осталось **Make Device** нашу модель, как и ранее с основными свойствами, т.е. **PRIMITIVE**, **MODDLL** и **TTRIGMIN** и начинаем добавлять сегменты. Поскольку их тут достаточно много, лучше это делать через заранее подготовленный текстовый файл. В папке вложения **8148_graphic** это файл **segments_8148.txt**. Поскольку у нас 72 светящихся элемента, получились 24 строки SEG, каждая из которых содержит записи для трех сегментов с разными координатами. Например, для **SEG1**, соответствующего в таблице сегментам **S1** и **S25** и **S49**, параметры будут выглядеть так

:

{SEG1=(3,400,200),(2,1800,200),(5,3900,200)}

Добавив все сегменты, проверяем себя с помощью теста. Во вложении **8148_graphic\Full_Test_Graphic.DSN** с помощью вспомогательной схемы этот процесс слегка «автоматизирован», чтобы можно было отследить последовательность зажигания сегментов в соответствии с таблицей. На этом этап подготовки графической модели индикатора окончен.

Теперь нам надо заняться моделью драйвера, имитирующего 2 последовательно соединенные **ML1001**. Для компоновки внутренней схемы двух-драйверного управления (назовем модель заранее **2xML1001**) я воспользуюсь модифицированными примитивами сдвиговых регистров и защелок. Мы уже ранее проводили такие вариации при моделировании АЦП (**п.6.16-6.18**). Здесь, существует еще более жесткое ограничение в 32 разряда. Но нам это не помеха, мне требуются примитивы на 24 разряда. Для большей компактности схемы дочернего листа я решил «ужать» входы/выходы разрядов в шины. В результате появились на свет следующие два примитива (Рис. 137).

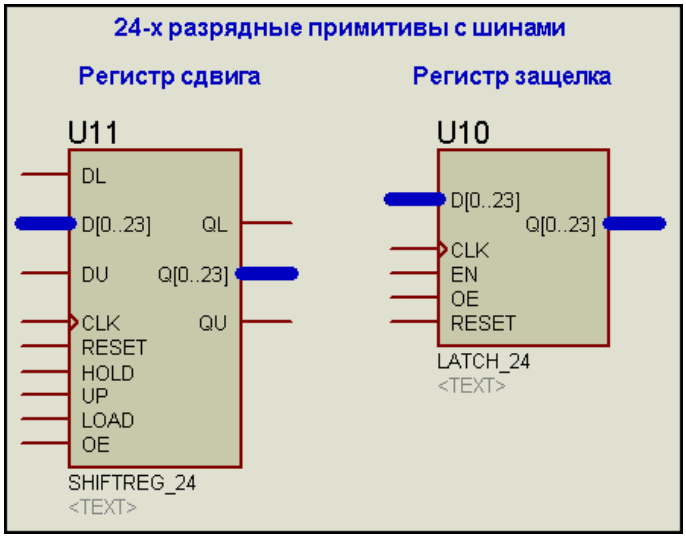


Рис. 137

Выглядят они более компактно, и схема дочернего листа будет иметь более опрятный вид. Надеюсь, что и полная картинка с дочернего листа в результате поместится непосредственно здесь. Теперь создаем графику самого сдвоенного драйвера. Тут тоже все просто. К изображению **ML1001** из предыдущего варианта у нас добавились только три управляющих выхода для выводов **COM1** и **COM2** дисплея (Рис. 138).

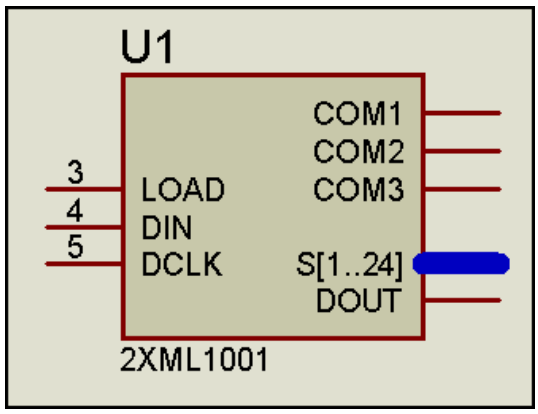


Рис. 138

Как обычно, присоединяем к нашей модели дочерний лист и с помощью вновь созданных примитивов формируем схему двух, соединенных последовательно драйверов (Рис 139). Эта структура отдельно приведена в папке вложения **2xML1001\Structure** проект **Int_Structure.DSN**.

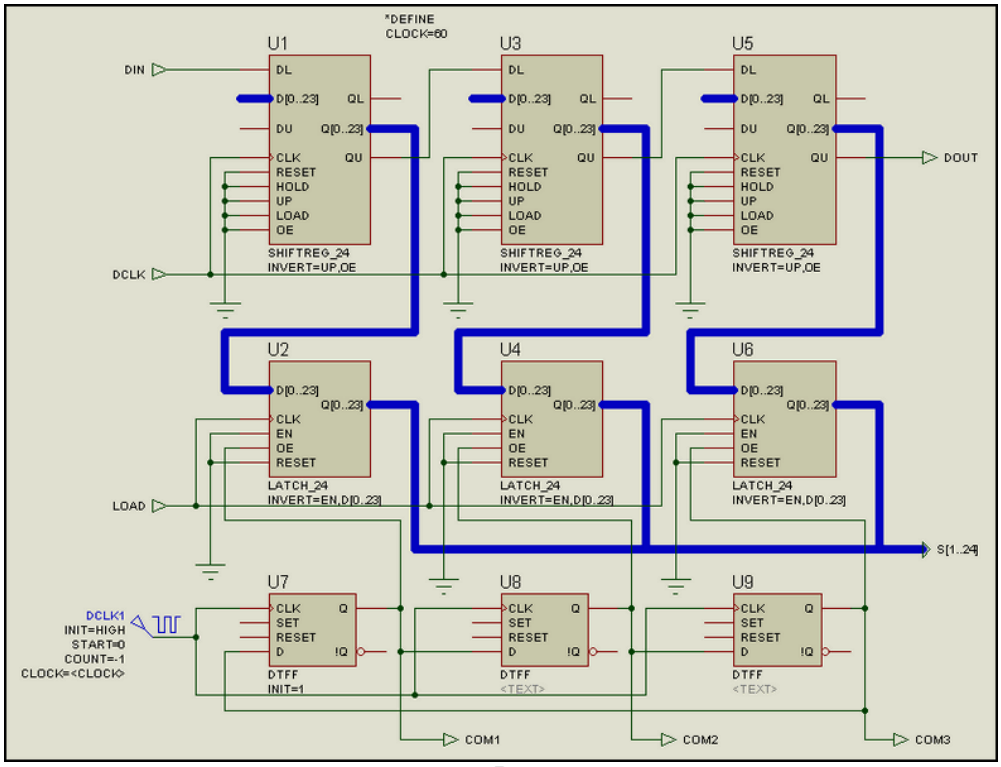


Рис. 139

Пожалуй, некоторые нюансы этой схемы заслуживают дополнительных пояснений. На регистрах сдвига **U1, U3, U5** организован последовательный сдвиговый регистр на 72 разряда. Чтобы не городить огород с завешиванием управляющих входов на шины с разными логическими уровнями, в свойствах входы **UP** и **OE** инвертированы – **INVERT=UP,OE**. Такой прием позволяет завешиванием всех управляющих входов примитива на землю сделать при этом инвертированные входы вечно активными. Аналогично в буферных регистрах защелках **U2, U4, U6** я поступил с входом **EN**. Так как выходные шины регистров сдвига соединены с входными защелок непосредственно, навешивание меток (лейблов) на них не требуется. Протеус и так поймет и соединит их «один в один», т.е. **Q0-D0**, **Q1-D1**, **Q3-D3** и т.д. Аналогично для выходов защелок и шинного терминала **S[1..24]** образуются пары **Q0-S1**, **Q1-S2**, но здесь есть одна хитрость, вернее две. Выходы защелок и шинный терминал должны иметь одинаковую разрядность, тогда образуются такие «сдвинутые» по нумерации пары. А еще выходы **U2, U4, U6** могут иметь третье (высокоимпедансное) состояние. Этим я воспользовался для коммутации выходов от трех регистров в общую шину. Напомню, что перевод в третье состояние осуществляется по входу **OE** (Output Enable). Вот его я и задействовал, чтобы на 24 выхода **S1…S24** подавать различные сигналы в зависимости от того, какой из выходов **COM** активен в данный момент. Еще нам потребуется какой-нибудь встроенный управляющий генератор. С этим тоже не проблема. Вспомните, как мы поступили при моделировании **К176ИЕ12**. Применяем тот же прием, ставим внутренний цифровой генератор **CLOCK**, а его параметры пропишем так, чтобы можно было задавать в свойствах модели. По умолчанию они прописаны в скрипте ***DEFINE** и равны 60 Герц. Этот генератор управляет трехразрядным кольцевым сдвиговым регистром на D-триггерах (**U7, U8, U9**), первый из них предустановлен в 1 (**INIT=1**). Этот регистр формирует выходные сигналы **COM**, а попутно и коммутирует буферные защелки. Тут я не стал мудрить и поставил обычные примитивы триггеров. Ну и еще пару слов о свойствах регистров защелок. Так как нам необходимо иметь активные нули для зажигания сегментов на выходах модели, инвертированы целиком входные шины защелок, т.е. в их свойствах стоит **INVERT=EN,D[0..23]** (про **EN** см. выше). В этом варианте можно было инвертировать только входы, т.к. у выходов используется третье состояние и если им задать INVERT, мы его потеряем. Чтобы исключить ложное «подмаргивание» сегментов в момент старта симуляции для защелок применена стартовая предустановка всех разрядов в единицы **INIT=0xFFFFFF**. Вот и все особенности внутренней структуры сдвоенного драйвера.

После тестирования (во вложении папка **With_Child_Sheet**) с дочернего листа этого проекта компилируем файл **2xML1001.MDF**. Тест окончательного варианта драйвера и готовый MDF лежат в папке **With_MDF_File**. Напомню, что в завершение работы над моделью драйвера необходимо прогнать **Make Device** для него еще раз и в свойствах на третьей вкладке прописать следующие:

MODFILE=2xML1001.MDF
CLOCK=60

Во вложении это уже сделано. Свойство **CLOCK** необходимо для того, чтобы можно было в **Edit Properties** сменить частоту внутреннего генератора, поэтому на третьей вкладке **Make Device** везде задаем ему **Normal**, ставим ограничение **Positive, None-Zero**, ну а в **Description** описываем его как-нибудь попонятнее, я, например, назвал **Internal Clock Generator**.

На этом я хочу завершить «показательные выступления» с **LCDMPX.DLL**. Охватить весь мировой ассортимент всевозможных LCD цифровых индикаторов, как вы понимаете, невозможно. Надеюсь, что приведенные примеры позволят вам при необходимости самостоятельно разработать требуемые для конкретного случая модели сегментных или мнемонических ЖК дисплеев. Первоначально я планировал сделать еще материал по **Holtek HT1611**, но, учитывая, что этот «раритет» практически исчез с рынка и если завалялся у кого, то только потому, что очень неудобен для использования в качестве индикатора – отсутствуют десятичные точки, я решил эту модель не делать. Если уж кому невмоготу, то по этой ссылке:

http://kazus.ru/forums/showpost.php?p=182152&postcount=518

лежит его модель от **Soir**. В свете новых знаний можете усовершенствовать этот вариант самостоятельно. Мы же переходим к моделям LCD дисплеев на основе контроллера **HD44780.**

К содержанию

8.14. LCDALFA.DLL – основа построения знакосинтезирующих дисплеев, базирующихся на контроллерах HD44780 и его клонах.

Пожалуй, дисплеи на базе **HD44780** – это наиболее востребованная и популярная у пользователей Протеуса линейка моделей ЖК индикации. На данный момент индикаторы этого типа выпускаются очень многими фирмами-производителями, как в «ближнем» зарубежье – КНР, так и в Европе, Америке и даже в России – МЭЛТ, а цены на наиболее простые, например, 8x2 без подсветки сопоставимы с ценами на сами микроконтроллеры, к которым они подключаются для Поэтому я решил уделить особенностям моделей на основе **LCDALFA.DLL** немного внимания, чтобы разобрать наиболее часто возникающие проблемы с симуляцией этих моделей, особенно у начинающих пользователей Протеуса.

Итак, все модели, базирующиеся на **LCDALFA.DLL**, расположены в двух библиотеках ISIS, а именно: **Optoelectronics\Alphanumeric LCDs** – типовые модели на базе **HD44780** и **Optoelectronics\Serial LCDs** – модели с последовательным вводом данных по одному проводу. Ну, что касается последних, то в России они практически не распространены, хотя любители экзотики могут заказать их через зарубежные Интернет-магазины. Достаточно набрать в любом поисковике фразу **Serial LCD Character Display** и вы получите массу ссылок на описание и предложения этих дисплеев, поскольку они достаточно популярны у западных разработчиков. Я же приведу здесь только одну ссылку: www.milinst.co.uk – сайт фирмы **Milford**, модели дисплеев которой и представлены в библиотеке **Serial LCDs**. А вот здесь: http://www.seetron.com/bpk000.html любители «экзотического секса» могут найти описание той самой дополнительно устанавливаемой платы **BPK** (от английского Backpack), которая вешается сзади на стандартный дисплей, чтобы он стал сериальным. И как это многочисленные наши «братья наши по разуму» до сих пор не содрали сей девайс, и не заполонили им наш рынок, особенно в свете развития проекта Arduino, где эти дисплеи очень популярны у западных разработчиков, ума не приложу. Но, хватит о грустном, вернемся к стандартным дисплеям на контроллере **HD44780**, которыми они нас снабжают в изрядном количестве. Наиболее популярная модель **LM016L** (2 строки по 16 символов) и окно ее свойств приведены на рисунке 140.

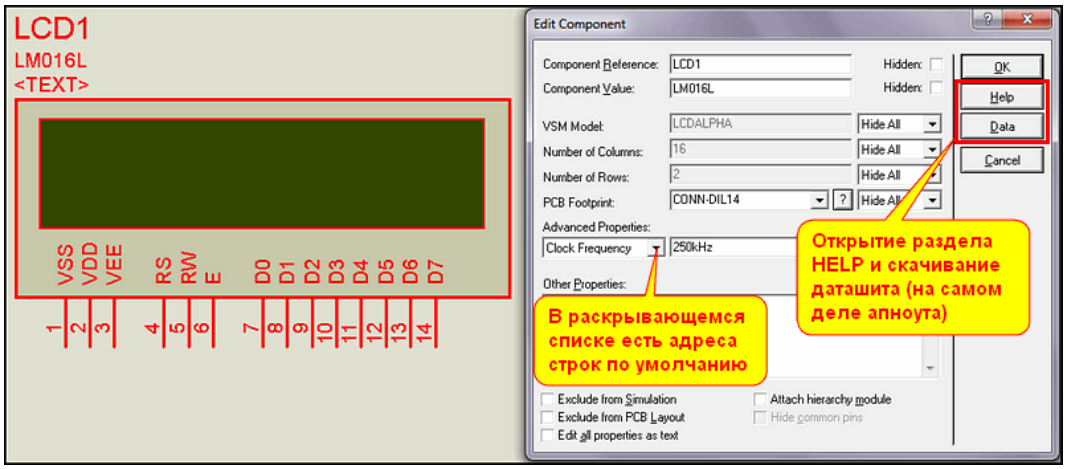


Рис. 140

Для начала немного о самой модели. Сразу же слегка «охолоним» начинающих, ножки **VSS**, **VDD** и **VEE** в данной модели приделаны просто, как недостающая часть табуретки. Модель прекрасно работает, даже если они просто висят в воздухе. Нет смысла, если конечно вы просто моделируете, а не разрабатываете печатную плату, в навешивании на них терминалов питания, каких либо батареек, выключателей и уж тем более регулирующих контрастность потенциометров. Кроме лишних «тормозов» при симуляции вы такими действиями ничего не получите, тем более, что регулировка контрастности в модели просто не реализована, ведь это чисто цифровая программная модель. Выводы **RS** (Register Selection), **RW** (Read/Write) и **E** (Enable), а также выводы 8-ми разрядной шины **D0…D7** в модели реализованы в полном объеме и используются при симуляции по прямому назначению в соответствии с даташитом.

Теперь заглянем в окно свойств модели на рисунке 140 справа. Здесь я хотел бы остановиться на двух моментах. В **Advanced Properties** по умолчанию фигурирует тактовая частота контроллера HD44780 **Clock Frequency=250kHz**. Это стандартная частота для конкретного контроллера от **Hitachi**, под которую и заданы все временные задержки модели в ISIS. В данном случае речь идет о внешней тактовой частоте обмена с контроллером, обозначенной в даташите как **External clock frequency**. Однако, не стоит забывать о том, какой именно контроллер стоит в том индикаторе, который вы собираетесь использовать. Так, в частности, не менее популярный клон **KS066U** от **Samsung** имеет стандартную **External Clock=270kHz**. Соответственно и выдержки времени при инициализации у него можно сделать несколько меньшими, хотя он и работает со стандартными для HD44780. Даташиты на наиболее распространенные **HD44780**, **KS066U** и **КБ1013ВГ6** есть во вложении. Последний, выпускаемый Зеленоградским НПО «Ангстрем», используется в популярных знакосинтезирующих ЖКИ **МЭЛТ**. Теперь поясню, почему я остановился подробно на этом моменте. Дело в том, что в моделях на основе **LCDALFA.DLL** заложен внутренний дебаггер (Рис. 141).

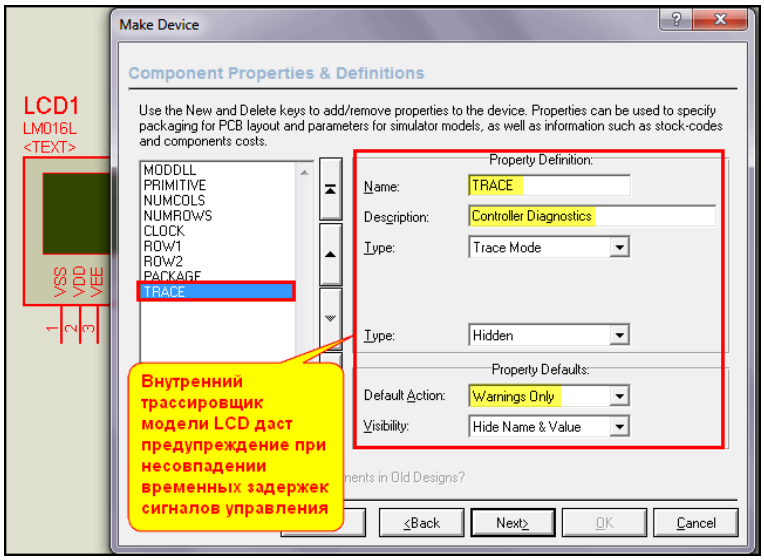


Рис. 141

В случае значительного отклонения во времени управляющих сигналов или отсутствия одного из таковых в программе инициализации дисплея в логе может появиться «горчичник», предупреждающий об этом, но далеко не всегда. Если с компиляторами с языков высокого уровня, где обычно присутствуют стандартные библиотеки для **HD44780**, такое случается очень редко, то использование в собственных разработках чужих ассемблерных программ инициализации ЖКИ, а уж тем более готовых HEX прошивок может надолго загнать вас в тупиковое положение. Причем, это совсем не значит, что прошивки не рабочие. Реальные дисплеи на основе данных контроллеров допускают некоторую вольность в обращении к ним. В результате программа, которая многократно повторялась в железе, работает в симуляторе криво, а иногда и вообще не работает. Но, сами понимаете, что написать такую математическую модель, которая будет автоматически предусматривать, что некто Вася или Петя решил подсократить выдержку при инициализации со стандартных для **HD44780** более чем 40ms до 12ms, потому что у него под рукой был русский МЭЛТ-овский дисплей практически невозможно, а уж тем более если часть команд инициализации вообще отсутсвует. К счастью, такие опусы легко распознаются в ISIS, даже при отсутствии исходника программы. До сих пор я скромно умалчивал о возможностях встроенного дебагера ISIS, но настала пора напомнить о нем, тем более, что именно в случае с ЖКИ может оказать существенную помощь.

Итак, сегодня в качестве «подопытного кролика» выступает разработка моего земляка А. Шарыпова почти десятилетней давности, которая, судя по многочисленным обсуждениям на

различных форумах, до сей поры не потеряла своей актуальности. Это опять частотомер и опять на **PIC16F84A**. Статья «Экономичный многофункциональный частотомер» была опубликована в октябрьском номере журнала «Радио» за 2002 год. Чтобы не искать, во вложении имеется DJVU вариант этой статьи. Прошивки с исходниками имеются на FTP сервере журнала, да и так их по сети разложено в изрядном количестве. Я буду рассматривать англоязычный вариант прошивки, но и в русском те же грабли. Собираем упрощенный вариант схемы и запускаем симуляцию с англоязычной прошивкой **counter.hex** (Рис. 142). Для сравнения в левом верхнем углу приведен вид дисплея из статьи. Как видим, вроде и работает, но нижняя строка полностью отсутствует. Причем, это наблюдается только в Протеусе, в железе вторая строка выводится. В чем же тут дело?

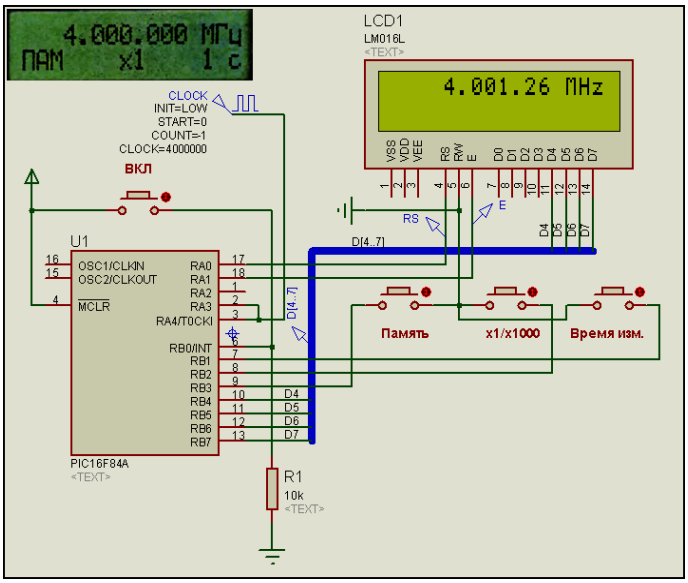


Рис. 142

Конечно, можно загнать наши сигналы обмена с контроллером дисплея в цифровой график и попытаться проанализировать по нему. Но есть более простой и наглядный метод. Переводим тот **TRACE**, что изображен на рисунке 141 из режима **Warning Only** (это те самые «горчичники»-предупреждения) в режим в режим **Debug** (отладка). Самый быстрый способ сделать это – щелкнуть правой кнопкой мышки по дисплею, выбрать **Configure Diagnostics** (фиолетовый паук) и перевести флажок для дисплея в нужное положение (Рис. 143).

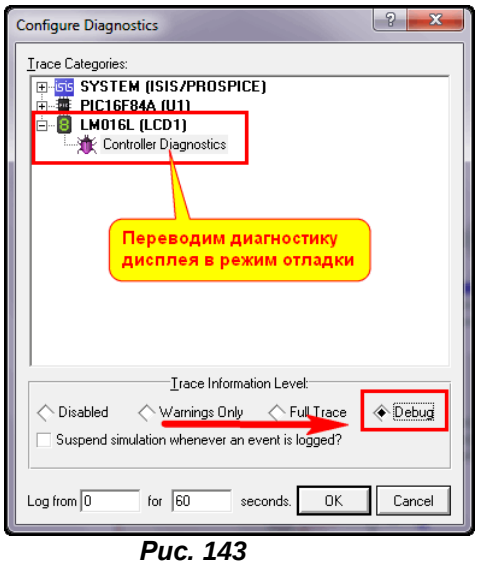


Рис. 143

После этого снова запускаем симуляцию и открываем окно **Simulation Log**, щелкнув левой кнопкой мышки справа от кнопок управления симуляцией. Результат приведен на рисунке 144. Те команды, которые относятся непосредственно к инициализации дисплея обведены рамкой. Не правда ли, скромный список по сравнению с тем, что приведен в даташитах на контроллеры для четырехбитного интерфейса. И как у нас может вывестись вторая строка, если двухстрочный режим даже не задан (**lines=1**). Самое интересное, что в железе это работает!!! Исправно выводится текст во вторую строку. Вот такой «запас прочности» у контроллера **HD44780**. Только вот вопрос – стоит

ли всегда надеяться на этот запас по любимому русскому принципу – «авось пронесет»? Вот и получается, что у автора работает, еще у 11 человек тоже, а тот пресловутый тринадцатый, у которого дисплей из «темного китайского подвала» остается в дураках.

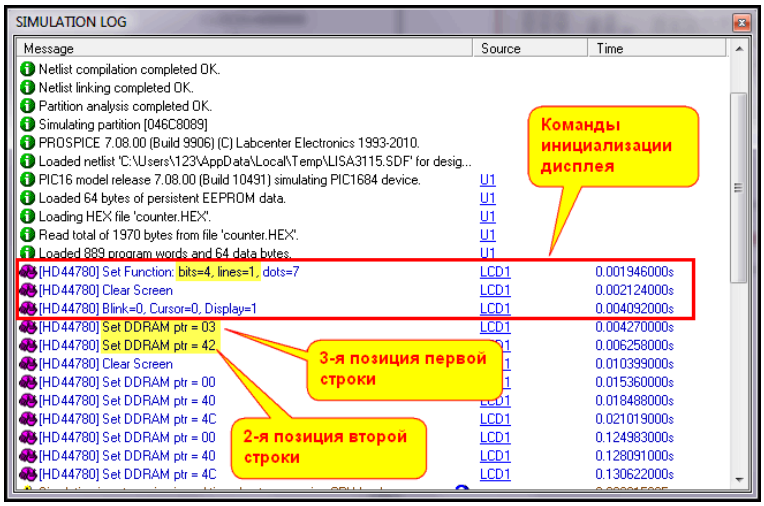


Рис. 144

Ну и законный вопрос – а можно ли в Протеусе увидеть нижнюю строку этого девайса? Конечно можно, но без кардинального вмешательства в программу – никак. На этот раз имеется исходник, будем надеяться, что я еще не совсем обленился и забыл ассемблер PIC. Мы не будем затрагивать основную часть программы, достаточно подкорректировать начальную инициализацию, а она выполняется единожды при подаче питания на девайс. Для начала заглянем в даташит. Я использовал русский от Ангстремовского контроллера, а красным цветом указал задержки из фирменного **HD44780** (Рис. 145).

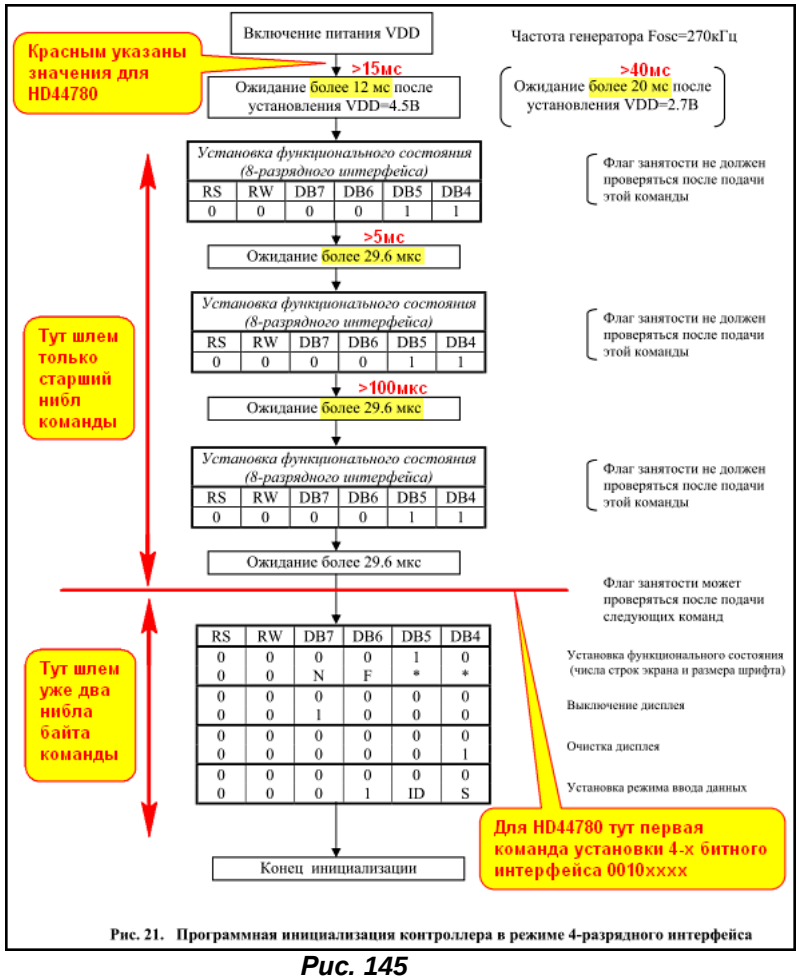


Рис. 145

Для нас значимыми в этой блок-схеме инициализации являются три задержки в 15 и 5 мс и в 100мкс. При подаче команд в 8-ми разрядном режиме. Напомню также, что команда передается при нулевом состоянии **RS** (**RW** у нас и так заземлен) и фиксируется положительным стробом на линии **E** контроллера ЖКИ. Программы инициализации для различных контроллеров несколько отличаются, но, поскольку модель в Протеусе сделана именно под **HD44780**, я буду руководствоваться его особенностями. Итак, покопавшись в ассемблерном листинге программы, и покрутив исходный вариант частотомера в **ISIS**, я обнаружил, что интервалы между стробами передачи команд и так достаточно большие около 90 мкс, что почти втрое превышает требуемые по даташиту. Начальную задержку в 15 мс я не стал делать, но желающие могут сформировать ее по аналогии с 5 мс из пачки следующих друг за другом **Pausem**. Для задержки в 5 мс используется существующая в программе **Pausem**. В авторском варианте она используется для формирования дополнительной задержки, необходимой, например, при подаче команды **Cursor Home** контроллеру ЖКИ. По даташиту она должна составлять 1,52 мс (в программе реально чуть больше – около 1,8 мс). Таким образом, три подряд вызова подпрограммы этой задержки дали мне требуемую около 5,4 мс. Несколько сложнее увеличено время между второй и третьей командой 8-ми разрядного интерфейса. В авторской программе имеется подпрограмма посылки строб-импульса по линии E - **Enbl**. Саму подпрограмму я использовал для передачи старшего нибла команд 8-ми разрядного интерфейса, а часть ее, поставив дополнительную метку **Pause25**, как подпрограмму задержки. В результате перед авторской инициализацией дисплея добавлен следующий ассемблерный код:

```
; ********** ДОБАВЛЕННЫЙ КОД ИНИЦИАЛИЗАЦИИ ЖКИ          **********
      movlw   0x30       ; Первая команда установки 8-ми битного интерфейса
      movwf          PortB       ; загружаем команду в порт B
      call  Enbl        ; Отсылаем старший нибл в ЖКИ
      call  Pausem      ; Три задержки Pausem в сумме ~5,5 мсек
      call  Pausem
      call  Pausem
; Порт B уже содержит команду 0x30 и не менялся
      call  Enbl        ; Вторая установка 8-ми битного интерфейса
      call  Pause25     ; В дополнительную задержку, чтобы  было >100 мксек
; Порт B по прежнему содержит команду 0x30 и не менялся
      call  Enbl        ; Третья установка 8-ми битного интерфейса
      movlw   0x20       ; Первая команда установки 4-ми битного интерфейса
      movwf          PortB       ; загружаем команду в порт B
      call  Enbl        ; Отсылаем старший нибл в ЖКИ
; Теперь мы уже в четырехбитном режиме и посылаем следующие команды
; через подпрограмму LEDcom двумя ниблами
      movlw   0x28      ; Установки 4 битн. интерфейс 2 строки 5x7 точек
      call  LEDcom
```

Надеюсь, тем, кто занимается программированием PIC-контроллеров на ассемблере, все в нем будет понятно из комментариев. Я постарался внести минимальные изменения в авторский вариант, только с целью добиться полной работоспособности частотомера в ISIS. Конечно, если посидеть подольше над программой можно было сделать и поопрятней и покороче, но я не стал с этим заморачиваться, конечный результат достигнут (Рис. 146).

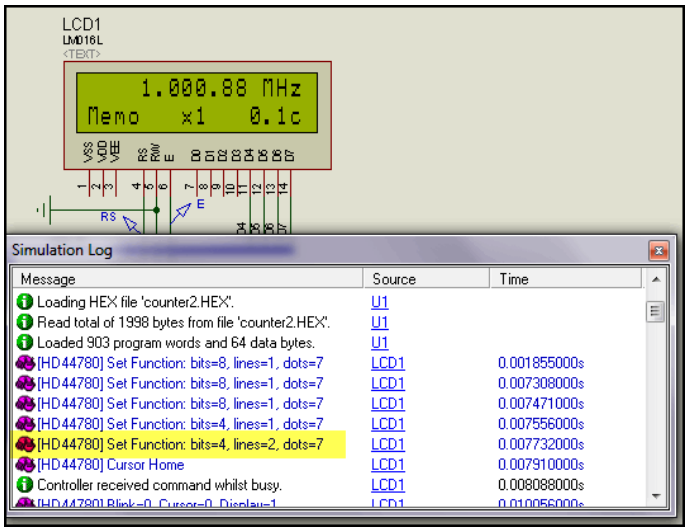


Рис. 146

Как видим, теперь начальная инициализация в логе стала выглядеть более приближенной к даташиту, а на индикаторе стала отображаться «утерянная» строка. Примеры данного частотомера

лежат в папке вложения **Sharypov** соответственно исходный вариант – папка **Eng_var** и откорректированный – **Eng_var_correct**. О том как использовать русскоязычный вариант в Протеусе чуть ниже.

В ходе поисков в интернете примеров для данной темы я натолкнулся на сайте Радиокота еще на один частотомер с ЖКИ на контроллере **PIC16F628A**. Поскольку там проблема индикации в Протеусе несколько другого плана, давайте рассмотрим и этот вариант.

Вот ссылка на саму тему у Кота: http://www.radiokot.ru/circuit/digital/measure/19/ , а здесь: http://www.radiokot.ru/forum/viewtopic.php?t=12555 страницы форума, посвященные обсуждению этой темы. Естественно, это все первоисточники, но данная схема уже благополучно расползлась по всемирной паутине и вы можете натолкнуться на нее и в других местах. Автор данного девайса тоже не особенно заморачивался, а просто переделал импортную программу, сделанную под более крутой контроллер на дешевый **PIC16F628A**. И конечно же он, и не только он, самостоятельно попробовали симуляцию программы в Протеусе. И тоже поймались на простейшей ошибке. Программа сделана изначально под однострочный LCD 16 символов. Если попытаться воспроизвести ее на однострочной модели LCD в ISIS, то половина информации будет отсутствовать, а на модели 16x2 эта информация появиться во второй строке. Тут все дело в некоторой путанице с адресацией для модели 16x1. Этот вариант дисплея нечто вроде «Оки» в линейке АвтоВАЗа, которую «любовно» именовали выкидышем. Дело в том, что это единственный производимый ЖКИ, у которого адресация в строке сделана не подряд, т.е. с первой по восьмую позицию адреса DD RAM **00-07**, а с девятой по шестнадцатую – **40-47**. Если посмотреть на адресацию двухстрочных ЖКИ, то последние совпадают с первыми восьмью второй строки. Отсюда и появляется информация на двухстрочном дисплее. Но здесь уже промашка допущена непосредственно в модели Протеуса и правится она в течение нескольких секунд. Кстати, на адресации в моделях ЖКИ на основе **LCDALFA.DLL** следует остановиться несколько подробнее, чтобы в дальнейшем она вас не смущала. Так, например, в даташитах популярных двухстрочных дисплеев 16x2 указаны адреса для первой строки – **00-0F**, для второй – **40-4F**. Для соответствующей модели **LM016L** в свойствах прописаны адреса **80-8F** и **C0-CF** соответственно (Рис. 147). В чем тут дело? Если внимательно посмотреть в даташит контроллера **HD44780**, то можно заметить, что установка старшего бита в единицу является признаком обращения к DD RAM, т.е. области памяти знакомест контроллера. Отнимите его из указанных для модели **LM016L** адресов и получится стандартная адресация. Теперь вернемся к однострочному ЖКИ на 16 символов, т.е. модели **LM020L**. В его свойствах для строки **Row 1** прописаны адреса **80-8F**, ну или в переводе на обычные – **00-0F** (подряд!!!). Это уже явный ляпсус Лабцентра, но помочь этому несложно, если заглянуть в Help модели. А там аглицким языком написано, что если адреса в строке идут не подряд, то они записываются диапазонами через пробел. Берем на вооружение данный факт и исправляем в свойствах следующим образом: **80-87 C0-C7**. Будьте внимательны! Если указанные диапазоны не совпадут с общим количеством символов в строке, ISIS при запуске симуляции пошлет вас в «эротическое путешествие» фразой красного цвета. В папке **RadioKot** вложения пример частотомера с 16-ти символьным однострочным ЖКИ и реальной индикацией частоты.

Ну и нам осталось разобраться только с модификацией **LCDALFA** под кириллицу. Изначально в библиотеке заложена таблица с европейским вариантом знакогенератора, а там, как видно из рисунка 147, русский алфавит отсутствует (эта часть таблицы выделена желтым).

Рис. 147

Таблица знакогенератора хранится в файле **LCDALFA.DLL** в виде точечного BMP рисунка размером 104x176 точек и извлечь ее оттуда и заменить на другую соответствующего размера можно любым редактором ресурсов (Restorator, PE Explorer, ResHacker и т.п.), а отредактировать даже в родном виндозном Paint. Но удобнее это сделать с помощью специализированной утилиты **charset.exe**, которая была написана нашим соотечественником, который присутствует на форуме под ником **Тень** и в данное время является штатным сотрудником **Labcenter Electronics**. Утилита довольно компактная и находится во вложении, а ее окошко приведено на рисунке 148. Пользоваться ей очень просто, с помощью кнопки открыть файл (3) открываете исходную **LCDALFA.DLL**, если есть необходимость – сохраняете исходный рисунок BMP с помощью кнопки (2). Затем подгружаете картинку с русским шрифтом с помощью кнопки (1) и сохраняете русифицированную DLL с помощью самой правой кнопки (4).

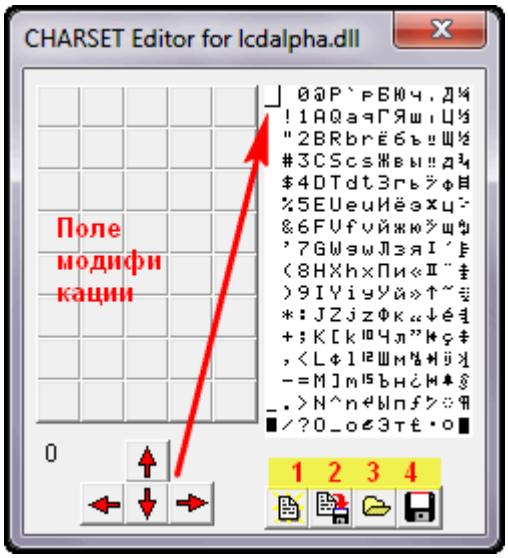


Рис. 148

На рисунке в окне программы уже загруженная таблица с русскими символами. Кроме того, можно модифицировать любой символ по своему усмотрению, выбрав его с помощью курсора (показан в левом верхнем углу таблицы). Курсор перемещается с помощью кнопок со стрелками, а выбранный символ отображается в поле модификации. Закрашенные пиксели в нем выглядят утопленными. На днях я проверил успешную работу утилиты на версии Протеуса 7.8. Единственный нюанс, под Windows 7 мне пришлось запустить ее от имени администратора. Ну и еще одна рекомендация – лучше проделать эту операцию в отдельной папке, а модифицированную DLL сохранить в папке **MODELS** под измененным именем, например, **LCDALFARUS.DLL**. Тогда можно использовать и тот и другой варианты, либо просто изменив **MODDLL** в свойствах модели LCD текущего проекта в режиме **Edit …as text**, либо продублировав все имеющиеся модели на базе HD44780 с добавкой RUS на конце и заменив для этих вариантов значение **MODDLL** на **LCDALFARUS.DLL**. В последнем случае, уже добавляя модель LCD в проект, выбираем нужный, например, **LM016** – западный, **LM016RUS** - с кириллицей. Готовый вариант **LCDALFARUS.DLL** для версии 7.8 во вложении.

На чем еще хотелось бы заострить ваше внимание. Некоторых смущает, что отсутствуют «хвосты» у символов русского шрифта. Например, если вы будете выводить на дисплей слово «Щука», то увидите «Шука». Тут просто уместно вспомнить, что стандартный вывод для дисплея по умолчанию 5x8 точек (при инициализации в 4-х битном режиме это команда **0x28**). Переведите дисплей при инициализации в режим 5x10 (заменив командой **0x2C**) и «хвост отрастет».

Из замеченных «глюков» для моделей на основе **LCDALFA** – это неадекватный по сравнению с даташитом сдвиг экрана аппаратной командой. Так что если планируете использовать дисплей для организации бегущей строки, то лучше это сделать программно или отлаживать сразу в железе, чтоб не напрягать себе мозги неожиданными «заморочками».

Ну, и напоследок немного о модификации. Нет, например, в библиотеках самой дешевой модели 2x8, а хочется. Все просто, разбиваем исходную **LM016**. Для графики есть одна особенность – маркер **ORIGIN** находится в центре, поэтому сдвигать (подрезать) экран и базу надо будет с двух сторон (Рис. 149).

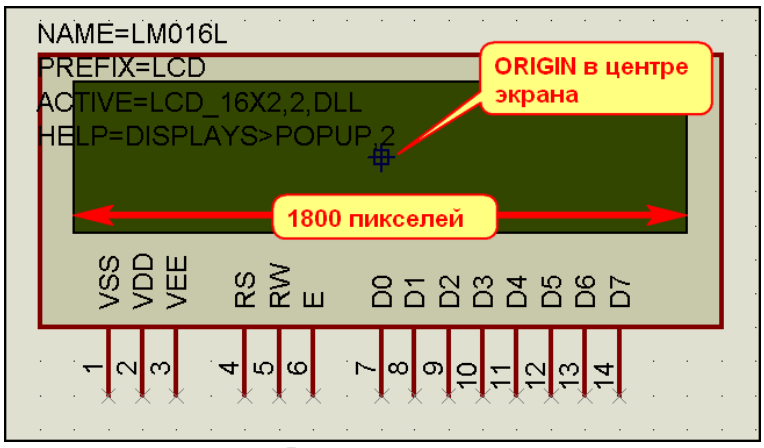


Рис. 149

Как ставить «фальшивку» маркера, вы уже знаете, и нетрудно определить, что ширина внутреннего зеленого экрана составляет 1800 пикселей (16 символов плюс два зазора по краям шириною как символ). Дальнейшее – арифметика для первоклассника. Изменить ширину придется как для основного изображения модели, так и символов потушенного и светящегося экрана, задав им соответствующие имена. Ну а при последующем **Make Device** необходимо на первой вкладке задать эти имена символов, а на третьей подкорректировать значение **NUMCOLS** (количество колонок) с 16 на 8 и задать новые конечные адреса для первой **ROW1** и второй **ROW2** строк. Кстати, можно и вообще не править графику, а поправить только адреса и количество колонок, ну и, конечно же, имя модели, но при этом пустые зазоры по краям экрана у вас будут широкими. Мы уже так часто проделывали подобные операции, что я пожалуй предоставлю вам самостоятельно поэкспериментировать с данными моделями, чтобы не надоедать лишними нудными подробностями. Дальше пойдет более интересный материал по моделям графических дисплеев, вот там и остановимся на тонкостях, тем более, что приемы редактирования будут похожими, а материал более актуален. Символьные же модели постепенно сдают позиции более популярным графическим ЖК индикаторам.

К содержанию

8.15. Обзор моделей контроллеров графических LCD, входящих в LCDPIXEL.DLL и моделей графических дисплеев на их основе. Особенности графических моделей и некоторые специфические параметры общие для всех моделей графических LCD.

Аналогично тому, как **LCDALFA** используется для создания знакосинтезирующих дисплеев на основе HD4470, библиотека **LCDPIXEL.DLL** служит основой для всех графических дисплеев, которые представлены в ISIS. Однако она объединяет функции не одного конкретного контроллера, а сразу нескольких контроллеров LCD с восьмиразрядной шиной данных. Рассмотрим кратко, какие контроллеры реализуются с помощью этой библиотеки.

T6963C (***Toshiba***) – один из самых «древних» контроллеров LCD, с помощью которого реализуются индикаторы с разрешением до 240x128 пикселей. Хотя данному контроллеру уже полтора десятка лет, дисплеи на его основе до сих пор выпускаются некоторыми производителями. В частности, на клоне этого контроллера **RA6993** тайваньской фирмы ***RAiO TECHNOLOGY INC*** www.raio.com.tw реализован ряд популярных у нас графических индикаторов фирмы ***Winstar*** http://www.winstar.com.tw/products_detail.php?CID=18&lang=ru серий WG и WB. Модель **T6963C** в ISIS не представлена в графическом виде в библиотеке **Optoelectronics/LCD Controllers**, так что о наименовании выводов и параметрах, задаваемых этой модели, можно судить только по представленным в библиотеках реальным моделям. На основе модели этого контроллера выполнены все модели ***Densitron***-овских индикаторов, начинающиеся с **LM** в библиотеке **Graphical LCDs**. Для них доступно скачивание англоязычных даташитов при нажатии кнопки **Data** в окне **Edit Properties**, поэтому подробно останавливаться на этих моделях я не стану. Английские даташит на **T6963C** и **Aplication Note** по использованию во вложении в папке Даташиты, а тех кто «хромает» в английском отправлю на http://www.gaw.ru/html.cgi/txt/lcd/chips/t6963/ , где в онлайн представлен русский перевод.

KS0108 (***Samsung***) – это тоже контроллер «долгожитель», в частности, документация самого Самсунга датируется 1997 годом. Несмотря на это, дисплеи с использованием данного драйвера выпускаются и в настоящее время, в том числе и Российской компанией МЭЛТ http://www.melt.com.ru/ , которая для этих дисплеев использует аналог **KS0108** производства ОАО «АНГСТРЕМ» – **К145ВГ10**. БИС драйвера LCD **KS0108** имеет встроенное ОЗУ матрицы и 64-канальный выход, который обычно используется для управления столбцами матрицы. Совместно с

KS0108, как правило, применяется дополнительно драйвер строк **KS0107**, который может управлять 64-мя строками матрицы. Таким образом, одна такая «сладкая парочка» способна обслуживать матрицу 64x64 точки, но чаще встречается комбинация из двух спаренных **KS0108** с разделенными выводами **CS**, которая применяется в дисплеях 128x64 пикселя. Это прародитель многочисленных клонов индикатор **AG-12864A** от **AMPIRE**, модель которого представлена в ISIS и называется **AMPIRE128X64**. Аналогичная модель, имеющая не инвертированные входы **CS**, представлена как **LGM12641BS1R**. Обе эти модели в библиотеках Протеуса отображаются как **Schematic**, поскольку для них имеется скомпилированный **KS0108X2.MDF** в котором и выполнено объединение двух контроллеров **KS0108**. Файл можно извлечь из **DISPLAY.LML** с помощью утилиты **GETMDF.EXE**. Об этом я рассказывал ранее. Модели контроллера **KS0108** и тех, о которых речь пойдет ниже, для создания подсхем имеются в библиотеке **Optoelectronics/LCD Controllers.** Англоязычный даташит на контроллер **KS0108** есть во вложении, а здесь: http://www.gaw.ru/html.cgi/txt/lcd/chips/ks0108b/index.htm находится русское описание в онлайн.

SED1520 (***Seiko Epson***) – это японское творение тоже благополучно здравствует с 1998 года. Один драйвер **SED1520** способен управлять матрицей 61x16 пикселей или двухстрочным знакосинтезирующим дисплеем формата 12x2, где знакоместо составляет 5x8 точек. На данный момент производителями дисплеев наиболее часто используется тандем из двух контроллеров **SED1520**, который обеспечивает управление матрицей графического LCD индикатора 122x32 точки. При этом память контроллера подразделяется на 4 страницы, обращение к которым происходит в зависимости от комбинации битов **D0**, **D1** в соответствующей команде **Set Page Address**. В библиотеках Протеуса представлен целый ряд моделей на основе данного драйвера: **AGM1232G *(AZ DISPLAYS INC)***, **EW12A03GLY *(EMERGING DISPLAY),*** **HDM32GS12-B** и **HDM32GS12Y-3 *(оба HANTRONIX)***. Даташиты на эти экзотические для России LCD доступны для скачивания при нажатии соответствующих кнопок в свойствах моделей. Для нас же больший интерес представляют дисплеи на основе **SED1520**, которые представлены на отечественном рынке электронных компонентов. В первую очередь это все та же компания МЭЛТ, которая выпускает графические дисплеи формата 122x32 на основе Ангстремовского клона этого драйвера – **КБ145ВГ4**. И он не единственный. Не менее распространен японский клон драйвера – **NJU6450 *(JRC*** http://www.njr.co.jp/english/index.html ***)***, а тот же **Winstar** выпускает индикаторы на базе тайваньского клона драйвера - **SBN1661G *(Avant Electronics*** http://www.avantsemi.com.tw ***)***. Чтобы не раздувать вложение даташитами до непомерных размеров, приведу только ссылку на русскоязычное описание **SED1520**: http://www.gaw.ru/html.cgi/txt/lcd/chips/sed1520/index.htm и оригинальный даташит от EPSON. Даташиты на клоны желающие могут скачать с сайтов производителей самостоятельно. В последний момент вспомнил про Минское **ОАО «ИНТЕГАЛ»**, которое также выпускает клон **NJU6450**, именуемый **IZ6450**, он по размеру не очень большой и на русском, поэтому добавлю во вложение.

SED1565 (***Seiko Epson***) – это еще один драйвер японской компании, модель которого представлена в ISIS. Контроллер обеспечивает управление матрицей до 65x132 и позволяет работать как с восьмиразрядной параллельной шиной данных, так и с последовательной загрузкой данных. В последнем случае в качестве входа данных **SI** используется пин **D7**, а в качестве входа тактового сигнала **SCL** пин **D6**. В Протеусе на основе драйвера **SED1565** реализован ряд моделей дисплеев формата 128x64 точек. Модели **HDG12864F-1 *(Hantronix)*** и **NOKIA7110** (96x65 точек) выполнены с последовательным интерфейсом. Для всех моделей дисплеев на базе **SED1565** возможно скачивание даташитов при нажатии соответствующей кнопки **DATA** в окне свойств. В данный момент почти нереально приобрести индикатор с контроллером данного типа, так как он не получил широкого распространения в отличие от предыдущих драйверов. Разве что кому то посчастливиться откопать такой раритет, как Нокия 7110 с исправным дисплеем. Соответственно и русского перевода документации по **SED1565** не существует, а английский вариант находится во вложении.

Теперь рассмотрим некоторые особенности построения моделей графических дисплеев, которые могут пригодиться нам в дальнейшем. Для начала немного о графике. Все дисплеи содержат серую подложку с выводами, экран и два символа экрана – погашенный (**_0** в конце имени) и с подсветкой (**_1** в конце имени) (Рис. 150). Маркер ORIGIN общего графического изображения и символов установлен в верхнем левом углу экрана, который у имеющихся в ISIS моделей выполнен либо в зеленых, либо в синих тонах.

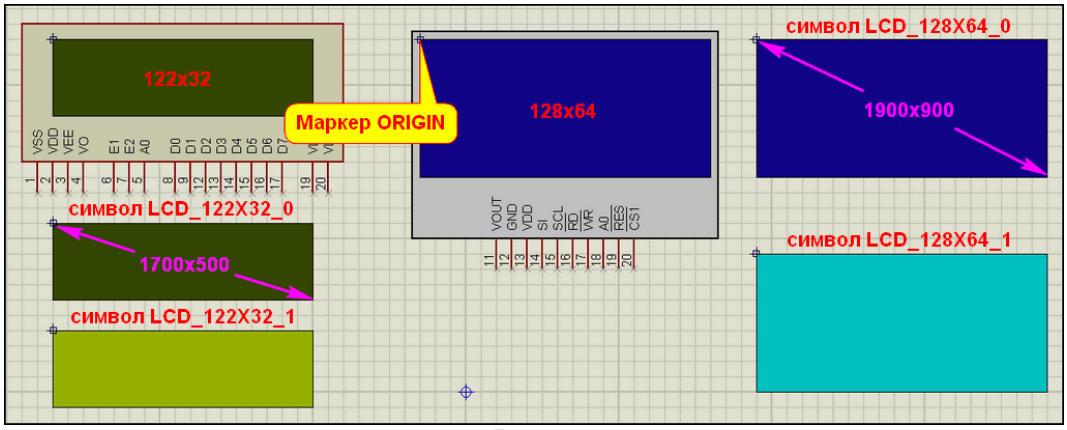


Рис. 150

Рабочую область светящегося экрана **LCDPIXEL.DLL** автоматически пропорционально делит на установленное разрешение, оставляя при этом небольшую незанятую окантовку. К примеру, на рисунке 150 слева приведен пример индикатора **EW12A03GLY** с форматом 122x32 точки. Символы погашенного **LCD_122X32_0** и подсвеченного **LCD_122X32_1** экранов составляют 1700x500 пикселей. Я «разобрал» (**Decompose**) эту модель, пропорционально увеличил вдвое ширину и длину символов, сохранив их с другим именем и конечно же увеличил и саму графическую модель. После этого сохранил ее с другим именем и подгрузил в типовой проект из примеров Протеуса **EW12A03GLY.DSN** из папки **SAMPLES\VSM for AVR\AVR and SED1520** версии 7.8. Что из этого получилось, видно на рисунке 151. Скриншот сделан в момент инверсного вывода, когда фон экрана воспроизводится черным. Здесь видна и нерабочая окантовка и рабочее поле экрана и самое главное – достигнутый эффект. Слева стандартная модель из библиотеки ISIS, справа моя модификация для себя-любимого, «слабовидящего и глухослышащего». Аналогично можно и уменьшить рабочую область дисплея. Пример из Протеуса с моими модификациями в папке **Sample_ModDLL** вложения. Надеюсь, общая концепция создания графики дисплеев ясна, пойдем дальше.

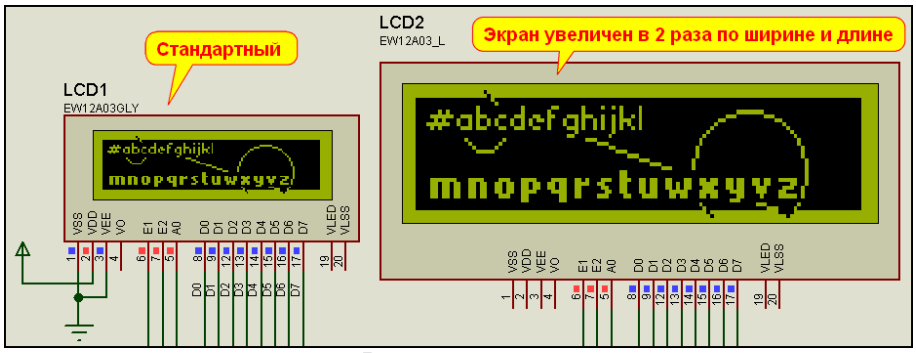


Рис. 151

Теперь немного разберемся с назначаемыми свойствами и параметрами.

MODDLL – **VSM Model DLL**. Ну, наверное, многие уже догадались, что на третьей вкладке **Make Device** этому свойству для всех графических дисплеев присвоено имя библиотеки **LCDPIXEL.DLL**. Кроме того, не забывайте на первой вкладке включать галочку **Link to DLL**, устанавливать количество символов **2** и прописывать основную часть имени символов погашенного и подсвеченного экрана для вашей модели.

MODFILE - **LISA Model File**. Этому свойству назначается MDF файл вашей модели, скомпилированный со схемы дочернего листа. **LCDPIXEL.DLL** поддерживает передачу параметров из MDF, что позволяет создавать модели с двумя и более контроллерами дисплеев. Именно поэтому большинство моделей позиционируются как **Schematic**. Т.е. на дочернем листе вы создаете внутреннюю структуру модели с использованием моделей контроллеров из библиотеки **LCD Controllers**, а также можете имитировать аналоговую часть, например нагрузку подсветки дисплея с помощью аналоговых примитивов. Мы этим займемся практически в следующем разделе.

WIDTH и **HEIGHT** – соответственно ширина и высота рабочего поля экрана индикатора в пикселях. Например, для модели **EW12A03GLY** они равны соответственно **122** и **32**. Когда я менял графику в примере чуть выше, я оставлял эти значения неизменными и симулятор пропорционально увеличивал или уменьшал размеры видимых точек в зависимости от размеров символов экрана моделей. Иными словами, они как бы задают симулятору коэффициент, на который надо поделить размер рабочей области модели, чтобы получить размер точки индикатора в реальных пикселях

экрана дисплея компьютера. (Вот это я загнул фразочку!!! Но, если пару раз перечитать, вроде понятно.)

TRACE - **Controller Diagnostics** (диагностические сообщения контроллера); **TRACE_CWR** - **Command Write Events** (сообщения при записи команд); **TRACE_MWR** - **Data Write Events** и **TRACE_MRD** - **Data Read Events** (соответственно сообщения при записи и чтении данных). Эти параметры относятся к диагностике и отладке модели и по умолчанию носят значения **Warning Only** (только предупреждения). С **TRACE** мы уже сталкивались при рассмотрении моделей на **HD44780**. Для моделей графических дисплеев добавлены еще три отладочных параметра, которые также можно перевести в режим **DEBUG** и использовать для вывода в лог симуляции дополнительных сообщений о процессе обмена с контроллером дисплея.

Конечно, у некоторых дисплеев есть и дополнительные параметры, назначение которых становится понятным только после детального изучения соответствующих даташитов. Рассматривать здесь каждый досконально я не имею ни времени, ни возможностей, поэтому на этом краткий обзор моделей завершается. А мы перейдем к практическому примеру и попробуем создать модель нашего отечественного графического дисплея, выпускаемого фирмой МЭЛТ на базе контроллера **SED1520**.

К содержанию

8.16. Графические LCD на основе контроллера SED1520 в ISIS и их особенности. Моделируем отечественные дисплеи МЭЛТ в Протеусе.

Незавершенный вариант, который позже будет дополнен практикой.

Выбор практического примера на **SED1520** выпал не случайно. Во-первых об этом просил один из участников форума, а во-вторых графические дисплеи на основе этого драйвера и его клонов имеют самую низкую стоимость и вполне доступны для приобретения. Конечно, существуют еще дисплеи от сотовых телефонов, которые можно купить еще дешевле, а иногда и просто выдернуть из старого ненужного телефона. Но в этом случае отлаживать устройство придется только в железе. А пайка шлейфов к дисплеям от сотовых, в которых обычно используются миниатюрные нестандартные разъемы, у начинающих может вызвать затруднения. Исключение составляет дисплей **Nokia 3310**, для которого модель Протеуса существует в природе, но об этом мы поговорим позже. А пока рассмотрим модель драйвера **SED1520** и как мы можем адаптировать ее под свои нужды. Все модели драйверов дисплеев находятся в библиотеке **Optoelectronics\LCD Controllers**. Конкретно модель **SED1520** и окно ее свойств показаны на рис 152. Даташит **SED1520** доступен для скачивания при наличии подключения к Интернет и нажатии соответствующей кнопки.

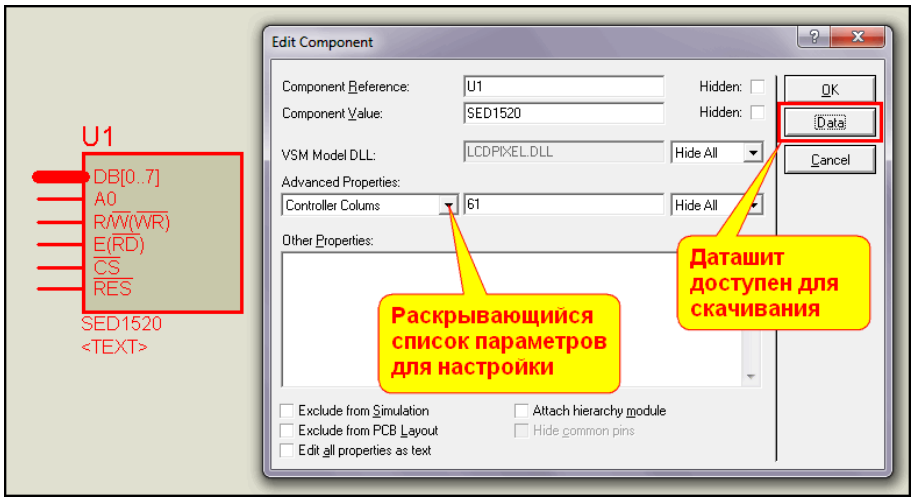


Рис. 152

Как видим, у модели имеются только выводы входов, а выходные сигналы колонок (**SEG0…SEG60**) и строк (**COM0…COM15**) уже жестко прописаны в программной модели Протеуса. Это и вызовет некоторое затруднение при моделировании **MT-12232A** (МЭЛТ), но оно вполне преодолимо с некоторыми условностями. Для начала немного информации о выводах модели из даташита. Сразу же напомню, что **SED1520** может работать в двух режимах: интерфейс с контроллером серии 68xx или интерфейс с контроллером серии 80xx. Нас интересует первый, поскольку он принят стандартом де-факто и для других микроконтроллеров (AVR, PIC). Для наглядности ниже синим цветом указано то, что относится к 68xx, а зеленым то, что относится к 80xx.

- **DB[0..7]** – двунаправленная восьмиразрядная шина команд/данных.
- **A0** – вход драйвера, определяющий что в данный момент передается по шине данных: **A0=0** – команда, **A0=1** – данные для вывода на индикатор.

- **R/W(WR)** – вход драйвера. Для 68xx определяет чтение (R/W=1) из **SED1520** или запись (R/W=0) в него. Для 80xx WR=0 при записи сигналы на шине данных стробируются положительным перепадом (передним фронтом) 0=>1 импульса на этом выводе.
- **E(RD)** – вход драйвера. Для 68xx определяет тактирование (выбор) данного драйвера. *(По E=1 производится запись/чтение в конкретный кристалл – это очень важный для нас сигнал.)* Для 80xx RD=0 означает, что шина D0…D7 **SED1520** направлена на вывод данных.
- **CS** – вход драйвера. Обычно CS=0. Эффективен при использовании внешнего генератора тактовой частоты.
- **RES** – вход сброса драйвера. Перепад сигнала на этом входе производит сброс микросхемы драйвера и установку для нее определенного интерфейса. Если был RES=0 и произошел переход 0=>1, то происходит сброс и устанавливается интерфейс 68xx. Если был RES=1 и произошел переход 1=>0, то происходит сброс и устанавливается интерфейс 80xx. *(Это тоже очень важный для нас сигнал.)*

Теперь заглянем в свойства модели **SED1520**, а конкретно в раскрывающийся список. С некоторыми параметрами оттуда мы уже знакомы, но в стандартном окне они не видны и мы сможем их увидеть только с установленной галочкой **Edit all properties as text** или в режиме **Make Device** на третьей вкладке. Итак, сначала о тех, что видны в раскрывающемся списке:

- **Controller Colums** *(да-да, именно **Colums**, а не **Columns** – и у англичан бывают грамматические ошибки)* – **CONTRWIDTH** по умолчанию равно 61 – количество колонок (**SEG**) в контроллере.
- **Controller Lines** **CONTRHEIGHT** по умолчанию равно 16 – количество строк (**COM**) для модели контроллера.
- **Controller Display X Offset** – **BMPXOFF** смещение картинки (на экране) по горизонтальной оси X. По умолчанию нулевое.
- **Controller Display Y Offset** – **BMPXOFF** смещение картинки (на экране) по вертикальной оси Y. По умолчанию нулевое.
- **Segments Output Direction** – **ADCMODE** направление вывода из памяти драйвера на экран. По умолчанию 0 – прямое, слева направо. При установке в 1 картинка выводится в обратном направлении справа-налево. Это вполне реальный параметр реального контроллера из даташита. Как мы увидим позже, именно с ним будут «заморочки» в МЭЛТ-овских индикаторах.

Остальные параметры имеют свойство **Hidden** (скрытые) и видны только при установке флажка **Edit all properties as text**, но среди них есть важные для нас и я их тоже опишу. В первую очередь это уже знакомые нам **WIDTH** (высота) и **HEIGHT** (ширина) но теперь уже экрана дисплея в пикселях. По умолчанию они соответственно 16 и 61. Также знакомы нам параметры трассировки: **TRACE**, **TRACE_CWR**, **TRACE_MWR** и **TRACE_MRD**. Все они по умолчанию **Warning Only** – режим предупреждений. Свойство **PRIMITIVE** для данной модели имеет значение **DIGITAL,SED1520**. Свойство **MODDLL** задано как **LCDPIXEL.DLL**. Надеюсь, эти свойства не нуждаются в особых комментариях. А вот со следующими двумя мы еще не встречались, поэтому коснусь их подробнее.

- **CTRLID** – идентификатор контроллера. По умолчанию равен **0x100**. Если в модели индикатора используются два или более контроллера, эти значения должны отличаться. Так у нас и будет в наших моделях – второму этот параметр мы присвоим как **0x101**.
- **RAMSIZE** – объем внутренней памяти драйвера в байтах. По умолчанию указан как **320** – это вполне реальное значение и его мы трогать не будем.

Ну, вроде с описанием модели **SED1520** пока все, пора приступать к реализации моделей индикаторов на нем. О построении графики моделей дисплеев мы уже говорили выше, но каждая модель графического индикатора имеет еще в свойствах и собственный MDF-файл, в котором реализована схематика соединения контроллеров. Вот о ней и пойдет речь. Для начала возьмем и воспроизведем по извлеченному из **DISPLAY.LML** файлу MDF схематику одной из существующих в ISIS моделей на базе **SED1520**, например, того же **EW12A03GLY**. Извлеченный MDF и воспроизведенная по нему схема находятся во вложении в папке **GLCD_recovery**. Эта же схема представлена на рисунке 153.

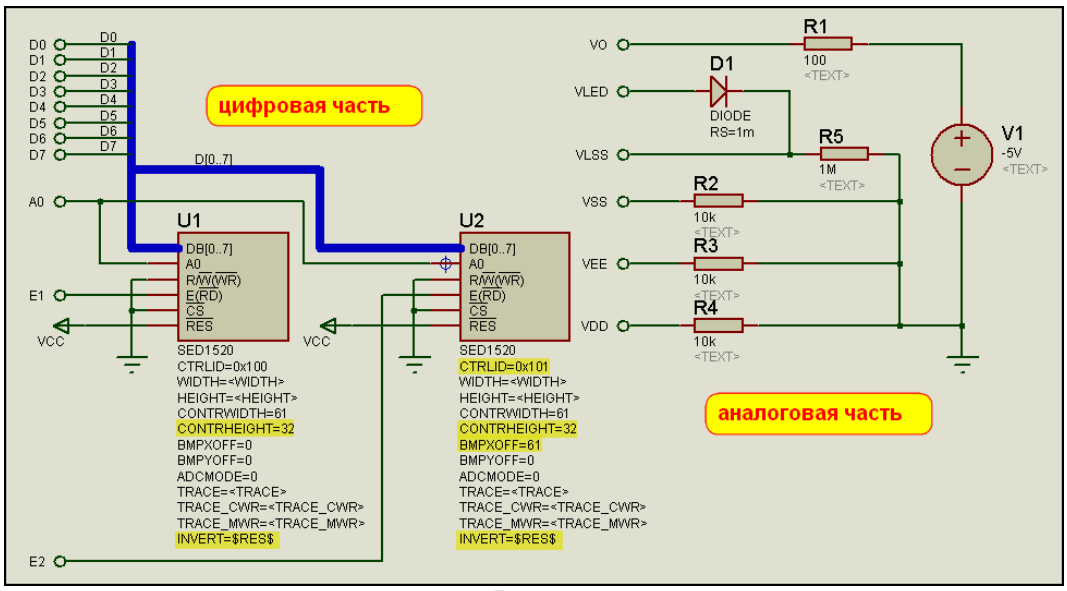


Рис. 153

Структуру модели индикатора **EW12A03GLY** можно условно разделить на две части: цифровую и аналоговую. Аналоговая часть имитирует нагрузки по выводам питания и подсветки и особенного интереса для нас не представляет, поскольку это в основном обычные резистивные нагрузки, за исключением диода, имитирующего светодиодную подсветку экрана и источника отрицательного напряжения, имитирующего встроенный в этот конкретный индикатор преобразователь напряжения. А вот на цифровой части остановимся отдельно. Мы видим, что в данном случае используются два контроллера **SED1520**, у которых большинство выводов объединено, а раздельными являются только выводы выбора **E1** для левого кристалла и **E2** для правого. Для данного конкретного индикатора выводы **R/W(WR)** завешены на землю, поскольку не производится чтение из контроллеров, а выводы сброса **RES** наоборот соединены с питанием и, кроме того, им присвоен параметр **INVERT=RES** для того, чтобы наш индикатор все время запускался в режиме интерфейса с МК серии 68xx.

Примечание. *Напомню, что **RES** ограниченное с двух сторон знаком доллара означает имя вывода со знаком надчеркивания (инверсный). При таких записях надо быть предельно внимательными, особенно, если вывод имеет длинное имя и только часть из него с надчеркиванием. Например, если вы захотите проинвертировать вывод **E** у **SED1520**, то необходимо писать его имя полностью со всеми скобками и т.п., т.е. **INVERT=E(RD)**. Иначе это свойство работать не будет.*

Выводы **CS** драйверов также завешены на землю, как и в большинстве реальных графических индикаторов на основе спаренных **SED1520**.

Какие же еще изменения внесены в свойства контроллеров прописанные по умолчанию. Как я уже и предупреждал ранее, для правого контроллера изменен **CTRLID=0x101**, чтобы кристаллы имели различные идентификаторы. Для обоих кристаллов количество строк **CONTRHEIGHT=32** вместо 16. Надеюсь, понятно почему, ведь это индикатор 122x32. По той же причине для правого кристалла, работающего на правую часть экрана индикатора, установлено горизонтальное смещение картинки по оси X на 61 пиксель - **BMPXOFF=61**. Еще хотелось бы обратить ваше внимание, что для ряда параметров – ширина и высота экрана, а также параметры отладки конкретные значения заменены на имена параметров в угловых скобках. Если кто-то подзабыл, напомню, что таким образом обеспечена возможность задания значений этим параметрам с основного родительского листа или точнее, в данном случае, из свойств графической модели (Рис. 154).

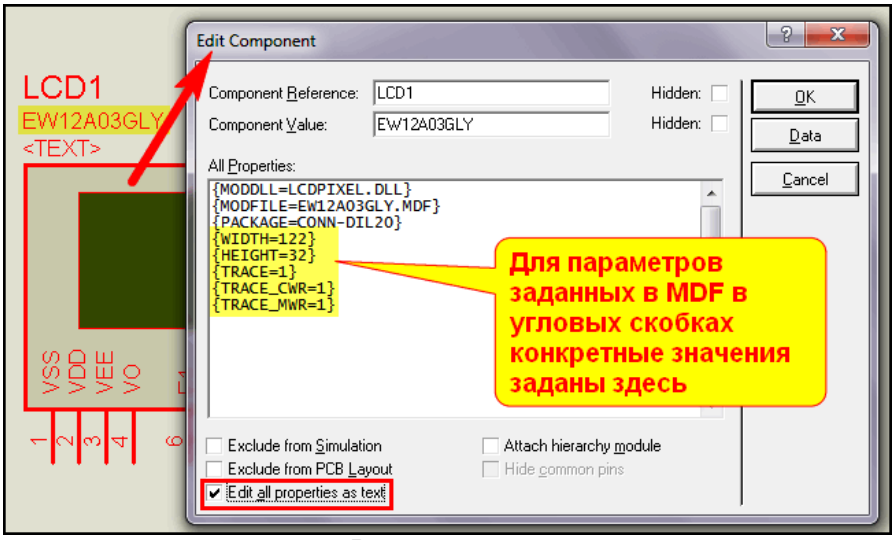


Рис. 154

Особенно хочу отметить, что не стоит пренебрегать заданием этим способом параметров трассировки (отладки) у сложных моделей, каковыми являются индикаторы. Если этого не сделать, то невозможно будет для уже скомпилированной модели включить режим отладки, а он очень важен для нас, в чем мы убедимся чуть ниже.

Ну вот, теперь надеюсь, всем стало ясно, почему программные **VSM** модели графических дисплеев прописаны как **Shematic** и имеют собственные MDF файлы. Я не стану больше останавливаться на зарубежных моделях LCD 122x32 на основе **SED1520**, поскольку в большинстве своем они построены почти одинаково с обязательным разнесением входов **E** для левого и правого кристаллов, т.е. у реального индикатора имеются входы **E1** для тактирования левого кристалла и **E2** для правого. Для примера на рисунке 155 диаграмма цикла записи из даташита **EW12A03GLY**, рассмотренного выше. Правда не знаю, зачем производитель этого индикатора приделал там и чтение, хотя и в даташите вывод **R/W(WR)** четко прорисован на землю.

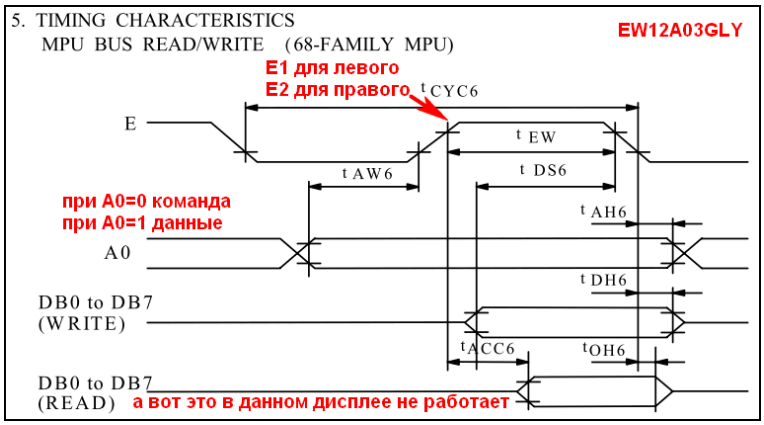


Рис. 155

Из диаграммы видно, что запись команды управления (при сигнале **A0=0**) или байта данных изображения (при сигнале **A0=1**) производится по положительному импульсу на входе **E1** в левый кристалл и на входе **E2** в правый. В других дисплеях возможно и чтение из кристаллов, но при этом учитывается логический уровень на входе **R/W(WR)**, который должен к моменту подачи тактового **E** при чтении быть в состоянии логической единицы, тогда будет верна нижняя часть диаграммы READ.

Ну а мы переходим к графическим индикаторам «местного пошива» и конкретно займемся дисплеями **MT-12232** компании МЭЛТ. Эти дисплеи формата 122x32 точки на основе Ангстремовского клона **КБ145ВГ4**, совместимого с **SED1520,** выпускаются с различными буквенными индексами, и уже тут скрывается первый «подводный камень». Дело в том, что индикатор **MT-12232B** среди остальных выделяется как белая ворона, поскольку выполнен по западным стандартам и полностью совместим по сигналам с большинством европейских и китайских дисплеев на базе **SED1520**. Он также имеет два раздельных входа тактирования **E1** и **E2** для разных кристаллов. Временная диаграмма для этого дисплея приведена на рисунке 156. Как видим, все отличие от предыдущей диаграммы в наличии сигнала **R/W(WR)**, обеспечивающего чтение из кристаллов.

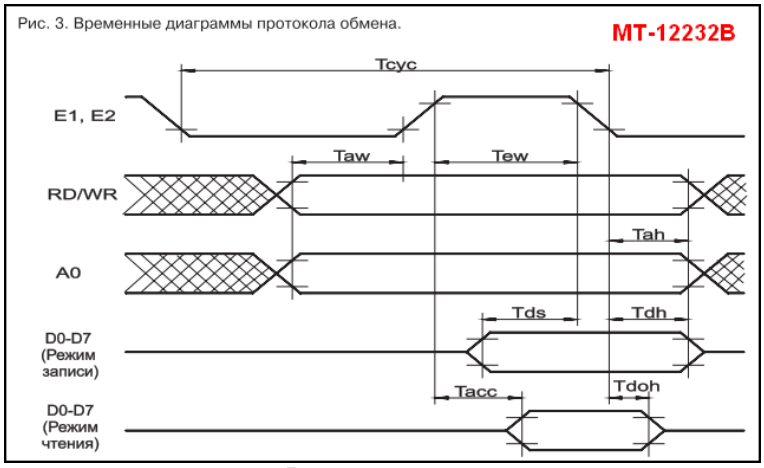


Рис. 156

Фактически для моделирования этого дисплея достаточно использовать уже существующую модель **AGM1232G**. Все отличие состоит в 3-м выводе контрастности **Vo**, который МЭЛТ не задействовал и обратной полярности выводов подсветки **BL** (19, 20). Поскольку ни то ни другое на процесс вывода данных на индикатор не участвует и реализовано только для имитации аналоговых свойств (нагрузки) эти выводы можно просто оставить «в воздухе». Ну а для тех, кому нужно соответствие «буква в букву» в папке **MT12232B** вложения находится графическая модель индикатора – поддиректория **Model_with_Child**. На дочернем листе **Model.DSN** находится переделанная под **MT12232B** подсхема **AGM1232G** (Рис. 157). Скомпилированный с нее файл модели **MT12232B.MDF** и тестовый проект находятся в подпапке **Test_MT12232B**.

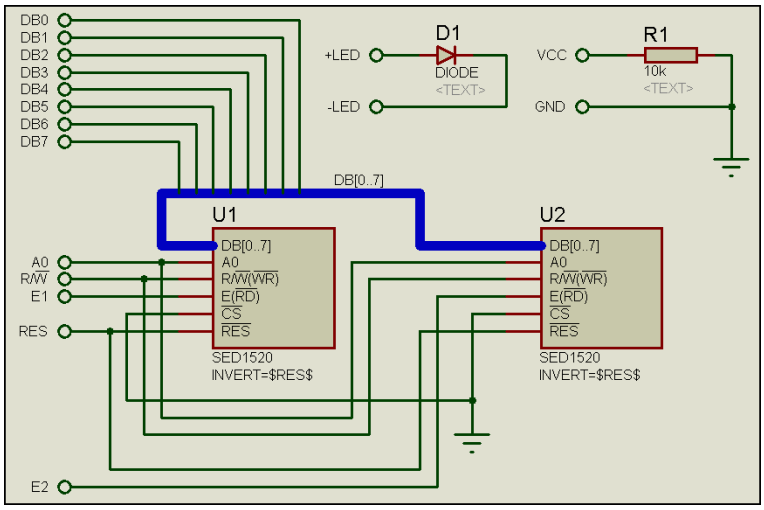


Рис. 157

Ну а мы далее на примере **MT12232A** рассмотрим особенности остальных индикаторов этой серии. Здесь МЭЛТ решил ввернуть свою «изюминку», которая проявляется в особенностях программирования и управления этими дисплеями. Не знаю, насколько это оправдано технологически, но с точки зрения моделирования здесь все становится с ног на голову.

Первая особенность дисплеев с индексами A, C, D и т.д. в отсутствии раздельных выводов **E** для управления кристаллами. Вывод **E** в этих индикаторах всего один и используется для тактирования как левого, так и правого контроллера **КБ145ВГ4**. Для обращения к конкретному кристаллу используется **CS**. Наличие лог. 1 на нем активирует обращение к левому драйверу, отвечающему за вывод информации в левую часть экрана. Лог. 0 на выводе **CS** означает работу с правым кристаллом, обслуживающим правую часть экрана.

Вторая особенность как раз и касается правого кристалла. Дело в том, что здесь МЭЛТ использовал еще и обратную последовательность подключения столбцов (колонок) драйвера к сегментам ЖКИ, т.е. выходу **SEG00** правого кристалла соответствует 122-я колонка дисплея, а выходу **SEG60** – 61-я колонка. Для нормального отображения картинки в правой части дисплея при начальной инициализации контроллеров необходимо для левого кристалла подать команду **ADC=0** (прямой вывод) а для правого **ADC=1** (обратный вывод) изображения. Эта особенность легко выполнима в реальной жизни, но при моделировании в Протеусе накладывает некоторые ограничения. Модель **SED1520** может воспроизводить данные в обратном порядке по команде

ADC=1, но выходы **SEG** мы «перепаять» задом наперед, как в реальном дисплее МЭЛТ не можем – они просто отсутствуют и жестко прописаны в программной модели. Поэтому при моделировании придется в программе инициализации для обоих кристаллов использовать **ADC=0**, а уже для реального «железного» дисплея перед прошивкой контроллера изменять это значение для одного из кристаллов в единицу. Как правило, инициализация проводится один раз при включении (запуске) устройства, поэтому очень больших проблем данный момент не вызывает. Главное – держать это на контроле и не забыть поменять значение при компиляции реальной прошивки.

Первая же особенность легко преодолима схемотехническим методом. Подсхема для компиляции MDF файла для **MT12232A** примет вид как на рисунке 158.

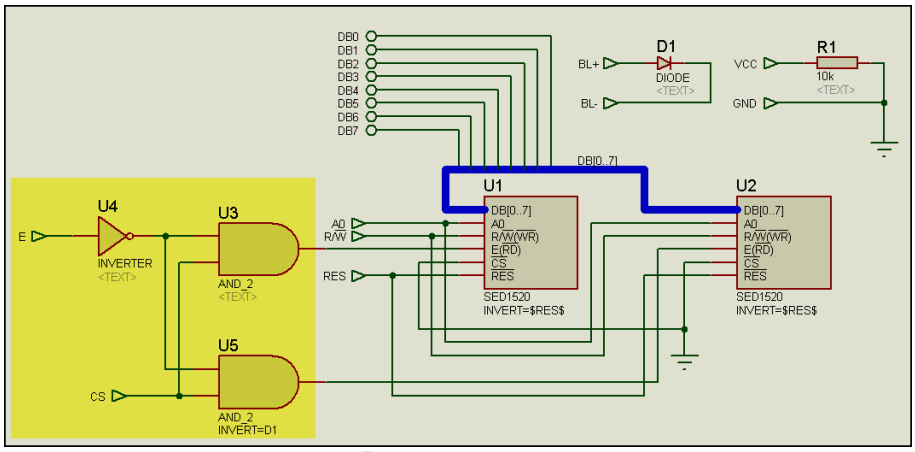


Рис. 158

Я специально оставил узел дешифрации сигнала **E** в первозданном виде, хотя можно было сделать и компактнее, убрав инвертор **U4** и задав свойство **INVERT** для входов **D0** элементов **U3** и **U5**. Вообще здесь уместны различные вариации по теме, но у меня заработало уже в таком виде, а большего и не надо. В папке вложения **MT12232A**, как и для модели с индексом B, расположены соответствующие **Model_with_Child** с дочерним листом, содержащим эту схему, и **Test_MT12232A**, в которой есть и готовый **MT12232A.MDF**.

Подводя итог этому материалу, хочу немного коснуться особенностей процедуры начальной инициализации и вывода информации на индикаторы МЭЛТ. В сети можно встретить различные варианты этой процедуры. В частности, в даташите на **MT12232A** производитель рекомендует следующую последовательность операций:

1. *после подачи напряжения питания удерживать вывод RES в состоянии логического “0” еще не менее 10 мкс;*
2. *подать перепад на вывод RES с логического “0” в логическую “1”, длительность фронта не более 10 мкс;*
3. *ожидать сброса бита RESET в байте состояния или выждать не менее 2 мс;*
4. *подать команду снятия флага RMW (END);*
5. *подать команду включения обычного режима работы (Static Drive ON/OFF);*
6. *подать команду выбора мультиплекса (Duty Select);*
7. *подать команду включения дисплея (Display ON/OFF).*

Первые три операции общие для обоих кристаллов дисплея и выполняются однократно. Они переводят дисплей в режим работы с МК 68xx. Остальные необходимо проделать для каждого кристалла в отдельности, причем в рекомендуемом производителем примере перед операцией 4 для каждого кристалла выдается еще и команда RESET (**0xE2**). Примеры программ от МЭЛТ находятся во вложении вместе с даташитами. Пункты алгоритма начальной установки, начиная с 4 можно переписать боле подробно следующим образом:

4. *подать команду (**A0=0, RD/WR=0**) сброса RESET (**DB7…DB0=0xE2**) в левый кристалл (**CS=0**), стробируем (1-0-1) сигналом **E**,*
   *подать команду (**A0=0, RD/WR=0**) сброса RESET (**DB7…DB0=0xE2**) в правый кристалл (**CS=1**), стробируем **E**.*
5. *подать команду (**A0=0, RD/WR=0**) сбросаRMW (**DB7…DB0=0xEE**) в левый кристалл (**CS=0**), стробируем **E**,*
   *подать команду (**A0=0, RD/WR=0**) сброса RMW (**DB7…DB0=0xEE**) в правый кристалл (**CS=1**), стробируем **E**.*
6. *подать команду (**A0=0, RD/WR=0**) нормальный режим (**DB7…DB0=0xA4**) в левый кристалл (**CS=0**), стробируем **E**,*
   *подать команду (**A0=0, RD/WR=0**) нормальный режим (**DB7…DB0=0xA4**) в правый кристалл (**CS=1**), стробируем **E**.*

7. *подать команду (**A0=0, RD/WR=0**) выбора мультиплекса 1/32 (**DB7…DB0=0xA9**) в левый кристалл (**CS=0**), стробируем **E**,*
   *подать команду (**A0=0, RD/WR=0**) выбора мультиплекса 1/32 (**DB7…DB0=0xA9**) в правый кристалл (**CS=1**), стробируем **E**.*
8. *подать команду (**A0=0, RD/WR=0**) установки верхней строки в 0 (**DB7…DB0=0xC0**) в левый кристалл (**CS=0**), стробируем **E**,*
   *подать команду (**A0=0, RD/WR=0**) установки верхней строки в 0 (**DB7…DB0=0xC0**) в правый кристалл (**CS=1**), стробируем **E**.*

9. *подать команду (**A0=0, RD/WR=0**) установки обратного соответствия (**DB7…DB0=0xA1**) (Внимание! Здесь для Протеуса должно быть **A0**, а для реального индикатора **A1**) в левый кристалл (**CS=0**), стробируем **E**,*
   *подать команду (**A0=0, RD/WR=0**) установки обратного соответствия (**DB7…DB0=0xA0**) в правый кристалл (**CS=1**), стробируем **E**.*
10. *подать команду (**A0=0, RD/WR=0**) включения дисплея (**DB7…DB0=0xAF**) в левый кристалл (**CS=0**), стробируем **E**,*
    *подать команду (**A0=0, RD/WR=0**) включения дисплея (**DB7…DB0=0xAF**) в правый кристалл (**CS=1**), стробируем **E**.*

Инициализацию можно выполнять как в той последовательности, которая указана выше, т.е. поочередно для каждого кристалла по одной команде, так и полностью все процедуры с 4-й по 10-ю сначала для одного (**CS=0**), а затем для другого (**CS=1**). Это уже зависит от того, кому как удобнее оформить программу. Необходимо только помнить про 9-й пункт, который в реальности будет отличаться.

К сожалению, все готовые библиотечные фунции на СИ для дисплеев **MT12232A**, найденные во всемирной паутине содержат те или иные ошибки. В основном это попытки подогнать стандартные функции вывода для дисплеев на базе **SED1520** под особенности **MT12232A**. Поэтому рекомендовать их для использования без внимательного изучения и коррекции я сейчас не могу. Но приведу в качестве примера вывод графического массива с логотипом МЭЛТ, который взят из примера производителя. В примере **LOGO.DSN** из папки **LOGO_MT12232A** вложения использован слегка модифицированный пример **MT12232-CV** приложенный к материалу о драйвере **MT12232A** с сайта **ChipEnable**:

http://www.chipenable.ru/index.php/how-connection/103-podkluchenie-mt12232-k-avr.html

К сожалению, рекомендовать полностью использовать данный материал для моделирования дисплея **MT12232A** в Протеусе не могу, но, во всяком случае, начальная инициализация и вывод графического массива на экран модели проходят корректно, что подтверждается примером (Рис. 159).

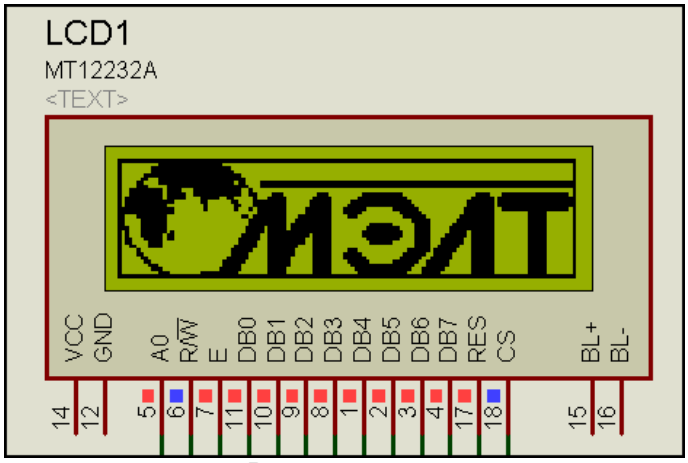


Рис. 159

Возможно, позже при наличии свободного времени я дополню этот материал новыми примерами, тем более, что сам индикатор у меня теперь есть в наличии и можно будет проверить соответствие модели «железу». А пока мы оставляем данный материал, подводим итог, поскольку уже давно не выкладывалась оффлайн версия материала, и переходим к рассмотрению активных моделей с элементами управления.

К содержанию