"Пианино" на ОСРВ

Введение

В данной статье рассматривается возможность обработки сенсорной клавиатуры с применением АЦП. В качестве примера разработаем программу «Пианино», обрабатывающую 36 сенсорных кнопок (3 октавы). Для интереса сделаем его многоголосым. В качестве аппаратной базы будем использовать демо-платы из набора pickit2 на базе контроллеров PIC16F690, PIC16F887 или PIC16F886.

Здесь 2-х минутное видео с демонстрацией того, что описывается в примере (пианист из меня, конечно, никакой).

Видео HQ (34 Mb)

<html> <object width=«640» height=«480»> <param name=«movie» value=«http://www.youtube.com/v/li7-orwG8F0&hl=en&fs=1&rel=0 [http://www.youtube.com/v/li7-orwG8F0&hl=en&fs=1&rel=0]»></param> <param name=«allowFullScreen» value=«true»></param>

<embed src=«http://www.youtube.com/v/li7-orwG8F0&hl=en&fs=1&rel=0 [http://www.youtube.com/v/li7-orwG8F0&hl=en&fs=1&rel=0]» type=«application/x-shockwave-flash» allowfullscreen=«true» width=«640» height=«480»></embed>

</object> </html>

Видео LQ (6.3 Mb)

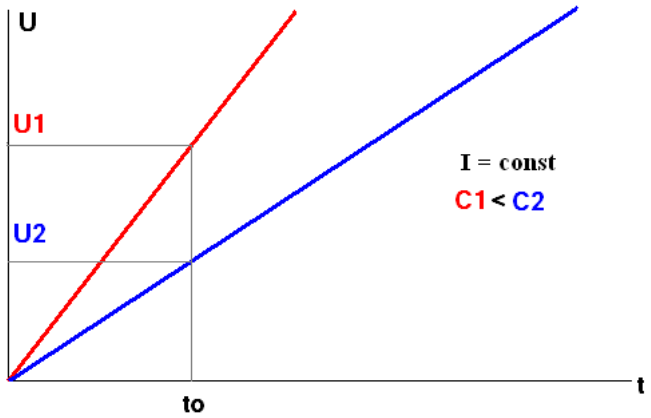
www.youtube.com/v/1oic1seP8u8

В опытном образце применен контроллер PIC16F88. Честно признаюсь: проверял только на нем, поскольку других под рукой не оказалось. По мере их доставания буду проверять. А если кто-нибудь соберет пианино и обнаружит, что что-то не работает, не стесняйтесь ругаться на osa@pic24.ru [mailto:osa@pic24.ru], будем исправлять.

Немного теории

Сенсорные кнопки

Основные принципы работы с сенсорными кнопками описаны **здесь**. Суть заключается в том, что когда мы прикасаемся пальцем к металлической пластине, мы вносим в схему дополнительную емкость. Это изменение емкости и фиксирует контроллер. Т.е. металлическая пластина является емкостным датчиком, представляющим собой конденсатор малой емкости. Если мы будем заряжать этот конденсатор через источник постоянного тока, то скорость нарастания напряжения на его обкладках будет пропорционально его емкости. Если измерение напряжения производить через одно и то же время после начала заряда конденсатора, то, очевидно, что при меньшем значении емкости конденсатор успеет зарядиться сильнее и, следовательно, напряжение на его обкладках будет выше.



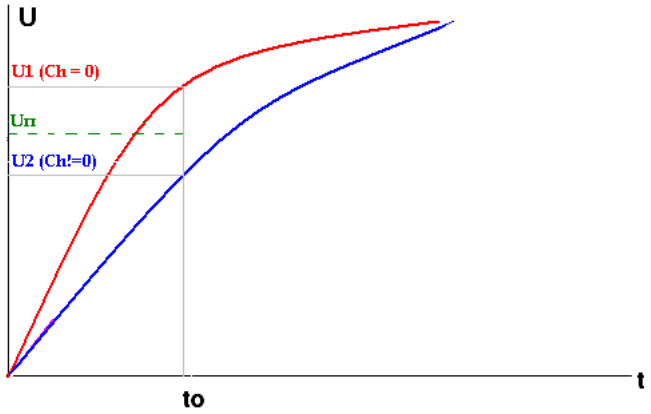
Два конденсатора с емкостями C1 и C2 (C1 < C2) одновременно начали заряжать одинаковым током. На графике видно, что в момент времени t0 напряжение на конденсаторе C1 успело вырасти больше, чем на C2. Итак, мы знаем, что при касании металлической пластины (емкостного датчика) пальцем, мы добавляем в схему емкость. Если мы будем периодически заряжать/разряжать конденсатор емкостного датчика и измерять напряжение на его обкладках через одно и то же время после начала заряда (t0), то мы будем получать всегда одно и то же значение (оно будет меняться от измерения к измерению, но в малых пределах). Если же мы коснемся емкостного датчика пальцем, то его емкость возрастет, и при измерении напряжения через время t0 мы обнаружим, что его значение немного меньше обычного.

Этот принцип, т.е. измерение напряжения на емкостном датчике через одинаковое время после начала его заряда, мы и будем использовать для чтения состояния сенсорных кнопок.

Практическая реализация

Итак, для начала нам нужен емкостной датчик. В качестве него нам подойдет любая металлическая пластина. Далеко ходить не будем, и воспользуемся примером из статьи по приведенной выше ссылке, а именно - используем в качестве пластины монету.

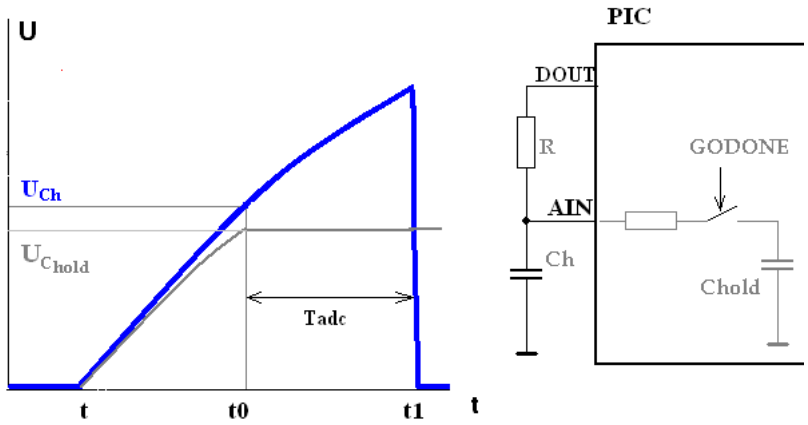
Теперь нам нужен источник тока. В идеале хотелось бы иметь источник постоянного тока (в контроллерах PIC24FJ256GA110 и PIC24FJ256GB106 со встроенным модулем CTMU делается именно так, т.е. заряд конденсатора в емкостном датчике производится постоянным током). При малом количестве кнопок можно было сделать именно так. Но у нас много кнопок (36), и такое решение было бы громоздким. Можно ли обойтись без источника постоянного тока? Можно ли использовать RC-цепочку для заряда конденсатора емкостного датчика? Рассмотрим рисунок:



Конденсаторы C1 и C2 (C1 < C2) стоят в RC-цепочках с одинаковыми значениями сопротивлений резисторов. Очевидно, что конденсатор C1 будет заряжаться быстрее, чем C2. И хоть графики заряда и нелинейные, мы все равно четко сможем определить, в каком случае емкость больше, т.к. через одинаковое время напряжения на конденсаторах будут разные. Для этого мы определим некий «порог срабатывания», заданный значением напряжения Uп. Так что мы можем позволить себе сделать допуск и вместо источника тока воспользоваться RC-цепочкой.

Далее для практической реализации нам нужно отмерять одинаковый временной интервал, после которого будет производиться измерение (t0). Здесь все достаточно просто: учитывая, что время t0 очень мало (единицы микросекунд), то задержку можно формировать программно пустым циклом. Единственное, о чем нельзя забывать, это запрет прерываний на время формирования задержки.

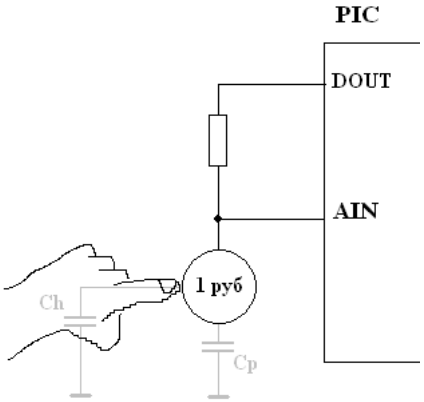
Наконец, последнее, что нам нужно сделать, - это измерение напряжения. Тут мы воспользуемся встроенным в контроллер АЦП. Рассмотрим рисунок:



Справа схема управления емкостным датчиком. Контроллер устанавливает выход DOUT в «1», чтобы начать заряжать конденсатор через задающий резистор. На входе AIN производится измерение напряжения. Чтобы конденсатор был всегда гарантированно разряжен, вывод контроллера AIN почти всегда настроен как цифровой выход, установленный в «0». Для снижения энергопотребления, чтобы, пока нет измерения, через резистор не тек ток, выход DOUT тоже устанавливается в «0». Когда нужно произвести измерение емкости конденсатора, мы делаем следующую последовательность действий:

1. Начинаем измерение в точке t. До этого момента напряжение на конденсаторе = 0, т.к. AIN настроена как цифровой выход, установленный в «0».
2. В момент времени t устанавливаем DOUT в «1», а AIN настраиваем как аналоговый вход.
3. Выдерживаем паузу до точки t0, чтобы дать конденсатору немного зарядиться.
4. В момент времени t0 начинаем АЦ-преобразование установкой бита GODONE. При этом внутри контроллера происходит «защелкивание» напряжения на внутреннем конденсаторе удержания Chold (график напряжения на нем показан серым цветом; скорость заряда Chold будет всегда чуть ниже из-за наличия последовательного сопротивления внутри контроллера).
5. В момент t1, когда преобразование закончено, устанавливаем вывод AIN на выход и записываем в него «0», чтобы разрядить конденсатор.
6. После этого вывод DOUT возвращается в «0».

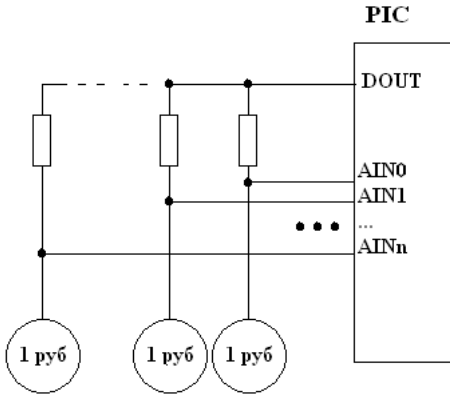
В результате для подключения одной кнопки мы имеем следующую схему:



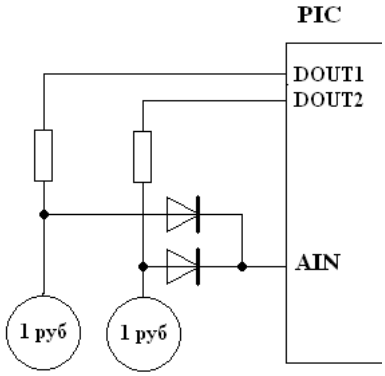
В качестве емкостного датчика выступает монета. Пока мы ее не трогаем пальцем, мы будем производить измерение емкости Cp, являющейся суммой паразитной емкости входа AIN, емкости между монетой и землей, емкостью между проводниками на плате (или проводами, подводящими монеты к плате). Когда мы касаемся монеты пальцем, мы добавляем еще емкость Ch (human), в результате чего измеряемая емкость увеличивается.

Много кнопок

По приведенному выше рисунку легко сделать вывод, что на каждый аналоговый вход контроллера можно повесить по монетке.

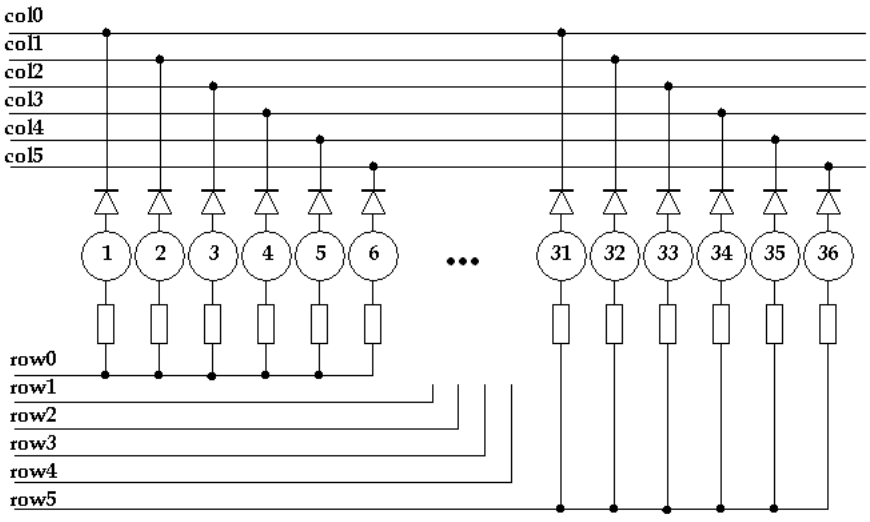


Но так мы получим максимум столько кнопок, сколько АЦП имеется на борту нашего контроллера (например, 12 у PIC16F690). А нам бы хотелось еще больше, т.к. пианино с одной октавой будет выглядеть довольно ущербно. Хотелось бы хотя бы 3 октавы (можно сделать и больше, но пока остановимся на трех). Нам нужно модифицировать схему так, чтобы один АЦП-вход имел возможность измерять напряжение на нескольких кнопках. Как же это сделать? Ответ прост: ставить разделяющие диоды.



Из рисунка видно, что, управляя цифровыми выходами DOUT1 и DOUT2, мы сможем входом AIN измерять отдельно емкость то одного, то другого емкостного датчика. Таким образом, на один аналоговый вход мы можем завести столько кнопок, сколько захотим. Ограничены мы лишь тем, сколько выводов контроллера мы сможем использовать как управляющие выходы DOUT. Комбинируя аналоговые входы и цифровые выходы, мы строим матрицу сенсорных кнопок. Очевидно, что сколько бы мы не выделили выводов под клавиатуру, максимальное количество кнопок получится, если количество аналоговых входов будет равно (или на 1 отличаться, если у нас нечетное количество выводов) количеству цифровых выходов. В рассмотренном примере «Пианино» для управления клавиатурой выделено 12 выводов, из них 6 - аналоговые входы и 6 - цифровые выходы. Получается матрица 6x6 = 36 кнопок.

В результате получаем следующую схему включения кнопок:

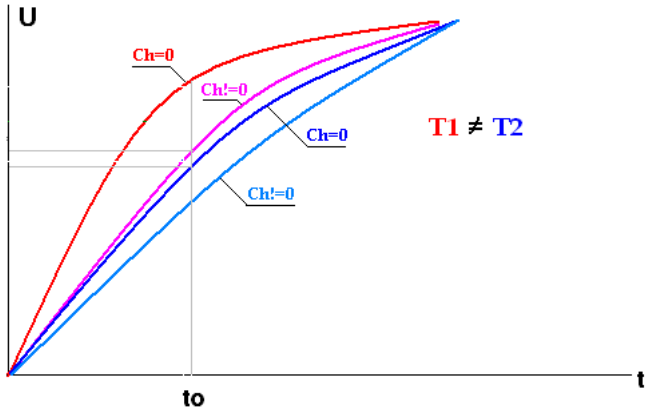


Примечание

Хоть описанный подход и был успешно реализован для обработки клавиатуры из 36 кнопок, он не лишен недостатков, которые следует предусматривать при проектировании устройства с использованием данного подхода.

Температурный дрейф RC-цепочки

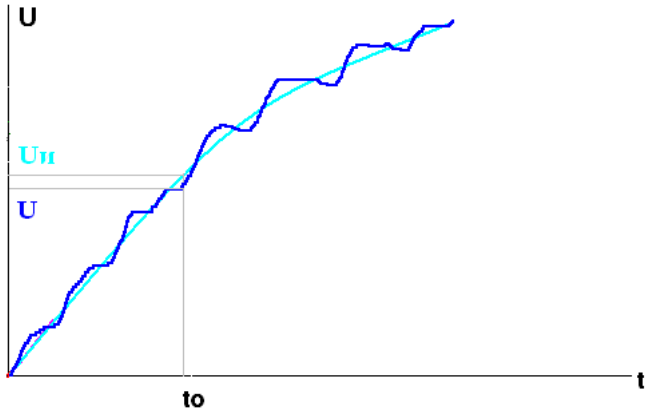
Сопротивление резистора в цепи емкостного датчика изменяется при изменении температуры. Так же меняется емкость внутреннего удерживающего конденсатора АЦП контроллера (Chold). Это приводит к тому, что крутизна графика зарядки емкостного датчика будет меняться с изменением температуры окружающей среды.



На рисунке показаны графики заряда RC-цепочки нажатой и не нажатой кнопки при сильно отличающихся температурах. Видно, что скорость заряда емкостного датчика без добавленной емкости Ch при температуре T2 приблизительно такая же, как и с добавлением Ch при температуре T1. Поэтому при проектировании устройств с емкостными сенсорными кнопками следует учитывать, что порог срабатывания кнопки нужно автоматически перенастраивать с каким-то интервалом (хотя бы раз в сутки).

Наводки при прикосновении

Когда мы касаемся пальцем металлической пластины, мы вносим в схему не только дополнительную емкость, но и источник помех. Поэтому в реальности график заряда Ch будет выглядеть так:



Голубым цветом изображен график без наводок, и для него напряжение в момент времени t0 будет равным Uи (идеальное). В реальности график будет искажен помехами от наводок (синий график). И реальное напряжение U в момент t0 будет всегда изменяться в некоторых пределах вокруг Uи, и оно может быть как больше Uи, так и меньше, в зависимости от фазы наводок в момент защелкивания напряжения на Chold. Эти наводки не будут нам мешать, если мы касаемся датчика непосредственно, т.к. в этом случае мы вносим в схему сравнительно большую емкость, и угол наклона графика изменится существенно. Но, если касаемся датчика через какую-то преграду (например, стекло или бумага), то вносимая в схему емкость будет мала и график отклонится хоть и заметно, но очень не сильно. А помехи будут его еще искажать, причем иногда так, что измеряемое напряжение может оказаться не только выше порогового, но и сильно приблизиться к измеренному без прикосновения напряжению, что затруднит определение факта нажатия кнопки.

Поэтому, если предполагается использование какой-то прокладки между емкостным датчиком и пальцем, то следует предусмотреть многократное считывание датчика и вычисление среднего значения. Это позволит более точно определить, было ли нажатие.

Паразитная емкость на входах

Все входы контроллера обладают некоей паразитной емкостью, которая также принимает участие в работе нашей RC-цепочки. Эта емкость - единицы пикофарад, однако в нашем случае ее влияние будет ощутимо. Проблема в том, что разные выводы контроллера имеют различные схемотехнические особенности: у них разная периферия, некоторые выводы имеют встроенный МОП-транзистор для обеспечения pull-up подтяжки (он даже в отключенном состоянии внесет свою толику в паразитную емкость) и т.п. Поэтому в программе следует предусмотреть то, что порог срабатывания на разных АЦП-входах будет различным.

Генерация звука

Генерация низкочастотным меандром

Самый распространенный способ генерации звука во встраиваемых системах - генерация меандра частоты, соответствующей частоте самого звука. Например, для генерации тона 1 КГц, мы формируем на выводе контроллера прямоугольный меандр частотой 1 КГц. В большинстве случаев такого способа вывода звука достаточно. Например, микроволновка сообщает нам, что разогрев окончен; брелок автомобильной сигнализации проигрывает мелодию при постановке/снятии с охраны; холодильник предупреждает нас, что мы забыли закрыть дверь и т.д.

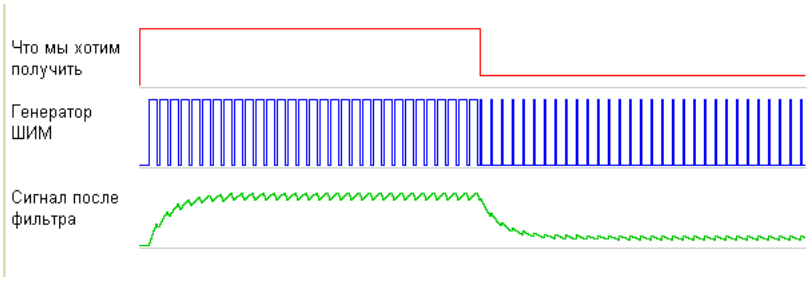
Такой способ прост, требует минимум ресурсов контроллера, достаточно информативен: можно давать звуки разной длительности, частоты, комбинации частот и пр. Можно ли таким способом получить многоголосье? В общем-то, можно, если на каждый звуковой канал выводить меандр своей частоты, а затем все эти меандры схематически суммировать и подавать на динамик. Однако тут есть несколько недостатков: во-первых, мы теряем несколько выводов микросхемы; во-вторых, сгенерировать два меандра разной частоты - это задача на порядок сложнее, чем сгенерировать один меандр (что сводит на нет преимущество в простоте реализации); наконец, в-третьих, звук получится очень неинтересным и даже немного раздражающим: из-за крутых фронтов прямоугольного сигнала звук будет очень резким.

Генерация с помощью ЦАП

Т.к. у PIC-контроллеров нет встроенного ЦАП-модуля, мы можем воспользоваться модулем ШИМ, работающим на высокой частоте. С его помощью мы сможем сгенерировать сигнал практически любой формы в заданном частотном диапазоне (частотный диапазон будет ограничен сверху в основном за счет частоты семплирования, об этом - ниже).

Генерация одноканального сигнала

Рассмотрим генерацию сигнала прямоугольной формы с помощью ШИМ.

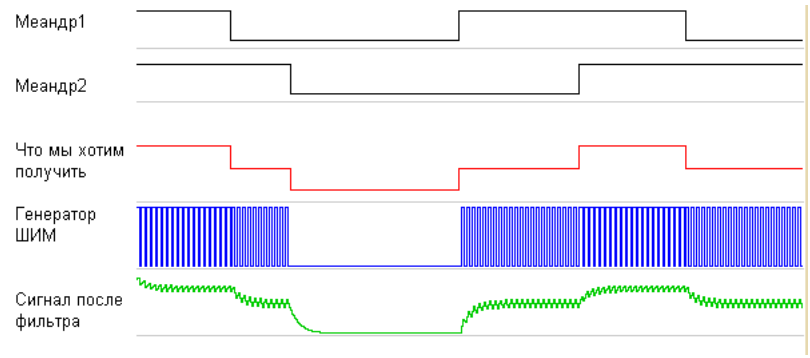


Когда мы генерируем «1», скважность импульсов ШИМ = 1 (на рисунке взята скважность чуть больше для наглядности, чтобы было видно, что частота ШИМ выше частоты генерируемого сигнала). Когда мы генерируем «0», скважность импульсов ШИМ максимальна (в идеале скважность равна бесконечности, т.е. импульсы отсутствуют, но на рисунке, опять же, для наглядности изображены просто импульсы большой скважности).

Пропустив цифровой сигнал с выхода ШИМ-модулятора через НЧ-фильтр, мы получим сигнал «прямоугольной формы» - зеленый график (в реальности он будет больше похож на прямоугольный, поскольку частота ШИМ гораздо выше приведенной на графике; по этой же причине пульсации будут гораздо меньше). В качестве НЧ-фильтра в самом простом случае может выступать RC-цепочка.

Генерация двухканального сигнала

Теперь перед нами стоит задача сгенерировать сразу два прямоугольных сигнала различной частоты, наложенных друг на друга. Когда мы генерировали один прямоугольный сигнал, мы приняли единичное состояние за максимальное значение (скважность импульсов ШИМ = 1), а нулевое - за минимальное (скважность = бесконечности). Но теперь нам нужно сгенерировать сигнал, равный сумме двух меандров, и за максимальное значение будет принято максимально возможное при таком суммировании, то есть то состояние, при котором и первый и второй сигналы находятся в «1». На рисунке схематически приведен пример использования ШИМ для генерации сразу двух меандров разной частоты.

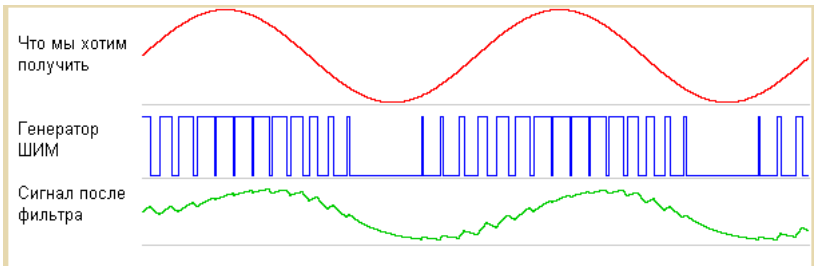


На рисунке видно, что на тех участках, где «1» установлена только на одном из каналов, напряжение на выходе фильтра равно половине напряжения питания (скважность импульсов ШИМ-сигнала на этих участках равна двум).

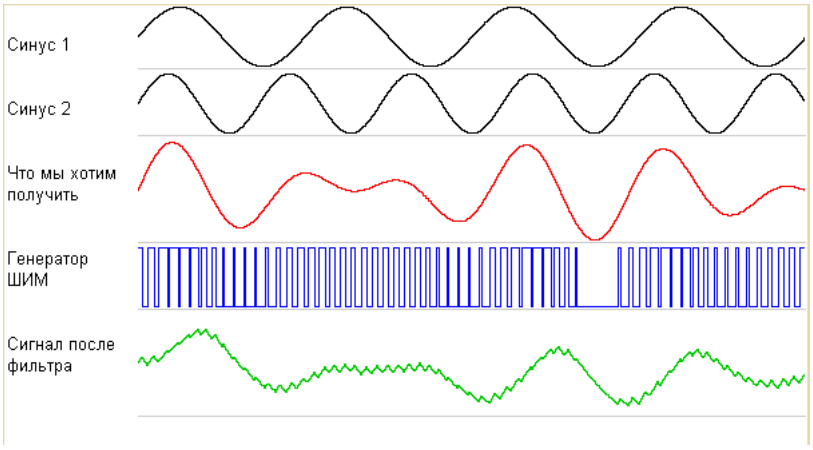
Теперь, когда стало ясно, как генерировать двухканальный звук, мы можем сгенерировать и трех- и четырех- и десятиканальный.

Генерация синуса

Генерация синусоидального сигнала, по сути, не отличается от генерации, описанной в предыдущем параграфе. Из графика синуса мы делаем точечные выборки с частотой, равной частоте ШИМ, и устанавливаем скважность текущего импульса соответствующей текущей точке выборки. Например, если мы взяли точку в самом пике синусоиды, то скважность будет равна 1; если мы взяли точку в середине периода (соответствующую 180 градусам), то скважность импульсов ШИМ-сигнала будет равна 2; если точка соответствует 45 градусам, то скважность будет равна (1 + 2^-2) ~ 2.41. Ниже приведен рисунок, показывающий генерацию синусоидального сигнала с помощью высокочастотной ШИМ:



Ну и, само собой, теперь для нас не проблема сформировать сигнал, являющийся суммой двух синусоид:



трех синусоид, пяти синусоид, двух синусоид и трех прямоугольников и т.д.

На этом введение в теорию закончим и приступим к реализации задуманного.

Проектирование программы

Зададимся задачей разработать программу «Пианино», которая будет способна воспроизводить до 8-ми нот одновременно и обрабатывать клавиатуру из 36 клавиш. Для разнообразия сделаем так, чтобы пианино могло синтезировать звуки различных тембров (тембр будет меняться нажатием кнопки).

Звук мы будет выводить через аппаратный ШИМ на максимально возможной частоте при разрешении 8 бит. При тактовой частоте контроллера, равной 20 МГц, максимальная частота 8-разрядного ШИМ будет 78 КГц. Частоту семплирования синтезатора выберем равной 20 КГц (хотелось бы выше, но 8 каналов звука быстрее не обработать).

Задачи

Сперва определимся с тем, какие задачи предстоит решать нашему контроллеру. Здесь же определимся с тем, какие из них оформим в виде задач ОСРВ, а какие поместим в прерывание.

1. Опрос клавиатуры - в этой задаче мы будем опрашивать 36 кнопок сенсорной клавиатуры по приведенной выше методике;
2. Генерация звука (синтезатор) - формирование скважности импульсов ШИМ в соответствие с нажатыми клавишами и выбранным инструментом.
3. Опрос кнопки - здесь будем ждать нажатия кнопки, и когда она будет нажата, будем выбирать новый музыкальный инструмент для синтезатора.

Задача генерации звука критична ко времени выполнения, т.к. у нас частота семплирования (т.е. изменения скважности ШИМ-сигнала) 20 КГц, то на один период выполнения задачи у нас максимум 50 мкс или 250 тактов (на самом деле еще меньше, т.к. нам нужно успевать и другие задачи обрабатывать). Есть смысл разбить эту задачу на две:

- первая будет заниматься исключительно формированием ШИМ-сигнала и будет помещена в прерывание (к ней и будет относиться требование уместиться в 250 тактов);
- вторая будет работать в фоновом режиме как обычная задача ОСРВ и будет заниматься формированием целеуказаний для первой в соответствие с нажатыми клавишами.

Опрос клавиатуры и опрос кнопки мы оформим в виде задач ОСРВ, т.к. они некритичны к скорости.

Итак, у нас получились 3 ОСРВ-задачи: «клавиатура», «кнопка» и «формирователь звуковых переменных» - и одна не ОСРВ-задача - «синтезатор», - которая будет помещена в прерывание.

"Клавиатура"

В функции этой задачи будут входить опрос всех 36 емкостных датчиков и формирование переменной состояния клавиатуры. Кроме того, задача должна будет как-то сообщать остальной программе, что состояние клавиатуры изменилось (либо что-то было нажато, либо что-то было отпущено).

"Кнопка"

Эта задача должна постоянно опрашивать состояние кнопки (с подавлением дребезга). При обнаружении факта нажатия кнопки должна будет происходить смена инструмента.

"формирователь данных для синтезатора"

Эта задача будет ожидать сообщения о нажатии/отпускании клавиш и формировать данные для синтезатора. Эти данные - по какому из восьми каналов генерируется звук для какой клавиши, или какой канал молчит.

"Синтезатор"

Для синтезирования звука в программе записан оцифрованный период звуковой волны (64 точки) в виде массива. Когда нужно генерировать звук, «синтезатор» при каждом запуске (1 раз в 50 мкс) выбирает из массива очередное значение и на основании него формирует скважность импульса для ШИМ-генератора. Массив оцифрованных значений звуковой волны рассматривается «синтезатором» как кольцевой, т.е. как бесконечный синус. Чем выше частота ноты, которую нам нужно синтезировать, тем с большим шагом выбираются значения из таблицы.

Когда нужно генерировать сразу два канала, из массива оцифрованного периода выбираются очередные значения для каждого канала по отдельности с шагами, соответствующими синтезируемым нотам. После этого прочитанные из таблицы значения суммируются, и на основании суммы формируется скважность импульсов ШИМ-генератора. Та же схема и для трех, четырех и т.д. каналов.

Мы предусмотрим в нашей программе синтезирование 4 различных инструментов, поэтому и массивов с данными об оцифрованных периодах в программе должно быть 4.

Данные

Здесь ответим себе на два вопроса:

- Какими данными будет оперировать наша программа?
- Какими данными и как будут обмениваться задачи?

Для клавиатуры

Нам потребуется некий массив для хранения информации о нажатых кнопках. Кнопок у нас 36, значит, это должен быть массив размерностью как минимум 5 байт. Далее, для индивидуальной установки/сброса бита для каждой клавиши нам нужна какая-то переменная для адресации конкретного бита в массиве. Проще всего для этой цели завести две переменные, одна из которых будет указывать на байт в массиве, а вторая будет являться маской бита в байте.

Кроме того, как уже было описано выше, пороги срабатывания клавиш могут меняться со временем, нам нужно иметь переменные для хранения порогов. И т.к. для разных АЦП-входов эти пороги могут различаться, то на каждый вход должна быть своя переменная, т.е. должен быть массив из 6 значений.

Все эти данные будут сведены в структуру:

```
struct
{
    unsigned char   Data[KBD_SIZE];         // Массив битов: 1 - кнопка нажата.
    unsigned char   cDataPos;               // Две переменные - указатель
    unsigned char   cDataMask;              // бита при приеме.
    unsigned char   Porogs[KBD_COLUMNS];    // Массив пороговых значений для
                                            // определения, нажата ли кнопка
} KBD;
```

Примечание. *Мы предусмотрительно пользуемся константами KBD_SIZE (=36) и KBD_COLUMNS (=6), оставляя себе возможность минимальными силами увеличить или сократить количество клавиш.*

Теперь, мы должны помнить, программа «Пианино» проектируется для разных контроллеров, а у разных контроллеров будут использованы разные выводы для матрицы клавиатуры. Поэтому нам нужно предусмотреть какой-нибудь удобный способ адресации этих выводов. Предположим, что в матрице в столбцах указываются аналоговые входы, а в строках - управляющие выходы. Рассмотрим, какие данные нам нужны для описания строк и столбцов. Со строками (управляющими выходами) все просто: нужны только адрес порта и маска бита в порту, по которой соответствующий вывод будет устанавливаться либо в «1», либо в «0». Со столбцами (аналоговыми входами), учитывая нашу методику, немного сложнее: нам нужен, во-первых, номер АЦП-канала, во-вторых, указатели на PORT и TRIS регистры, поскольку нам придется управлять и тем и другим, и, наконец, в-третьих, - маска бита в порту. Таким образом, мы формируем две структуры для шаблонов:

```
typedef struct          // Тип для задания аналогового входа
{                       //
    char  cADCChannel;  // Номер аналогового канала
    char *pPort;        // Указатель на регистр PORT
    char *pTris;        // Указатель на регистр направления
    char  cMask;        // Маска бита в порту
} TColumn;

typedef struct          // Тип для управляющего выхода
{                       //
    char *pPort;        // Указатель на регистр PORT
    char  cMask;        // Маска бита в порту
} TRow;
```

Теперь через эти типы можно объявлять массивы выводов контроллера, образующих строки и столбцы матрицы кнопок.

Примечание. *Учитывая, что обе переменные из типа TRow присутствуют в типе TColumn, можно было бы обойтись одним типом, просто для строк поля cADCChannel и pTris заполнять нулями, но для строгости мы будем использовать различные типы.*

Для синтезатора

Синтезатору для работы потребуется массив значений оцифрованных периодов синусоид для различных инструментов. Эти массивы будут храниться в программной памяти в виде констант (см. файл **sinus.c**). Т.к. скорость работы самого синтезатора хотелось бы максимально увеличить, то воспользуемся той особенностью, что обращение к массиву в RAM происходит быстрее, чем обращение к массиву в ROM. Поэтому для работы с массивом значений оцифрованного периода данные для текущего инструмента мы будем копировать в RAM. Т.е. нам нужно зарезервировать в RAM-памяти массив для этих целей:

```
char Sample[64];
```

Т.к. у нас предусмотрено синтезирование четырех инструментов, то в программе должна быть переменная, показывающая номер текущего инструмента. За выбор инструмента отвечает задача «Кнопка», в которой мы и будем производить копирование из ROM в RAM. Есть смысл сделать переменную, обозначающую номер текущего инструмента, статической внутри этой задачи:

```
static char s_cCurSample;
```

Теперь, синтезатор должен знать, какой канал «молчит», а по какому воспроизводится звук, причем ему нужно указать и частоту звука, и текущую фазу. Итак, у нас получается структура:

```
typedef struct          // Для переменных управления звуком
{
    unsigned int F;     // Частота
    unsigned int f;     // Фаза
    unsigned char key;  // Клавиша, которая проигрывается в данный момент. (0 - молчит)
} TSound;
```

Реализация

Кнопка

Как мы уже писали, по нажатию кнопки должен изменяться инструмент. Не будем забывать, что пользователь может и не менять инструмент, т.е. кнопку вообще не будет трогать, следовательно, какой-то инструмент нужно загрузить в самом начале.

```
void Task_Button (void)
{
    static char s_cCurSample;

    s_cCurSample = 0;

    CopySample(s_cCurSample);
    for (;;)
    {
        //--------------------------------------------------------------------------
        //  Ожидаем нажатия на кнопку (с устранением дребезга)
        //--------------------------------------------------------------------------

        do
        {
            OS_Cond_Wait(!pin_BUTTON);
            OS_Stimer_Delay(ST_BUTTON, 40 ms);
        } while (pin_BUTTON);

        //--------------------------------------------------------------------------
        //  Изменяем номер инструмента и копируем данные об инструменте в
        //  массив Sample
        //--------------------------------------------------------------------------
```

```
        s_cCurSample++;
        s_cCurSample &= 3;
        CopySample(s_cCurSample);

        //--------------------------------------------------------------------------
        //  Ждем отпускания кнопки
        //--------------------------------------------------------------------------

        do
        {
            OS_Cond_Wait(pin_BUTTON);
            OS_Stimer_Delay(ST_BUTTON, 40 ms);
        } while (!pin_BUTTON);


    }
}
```

Обратим внимание на то, что для формирования задержек используется не таймер задач (т.е. не сервис OS_Delay), а статический таймер. Это связано с тем, что код синтезатора, находящийся в прерывании вместе с системным обработчиком таймеров, может долго выполняться, когда активны все 8 каналов. И так как нам гарантированно нужно вместиться в 50 мкс (250 тактов), то любое ускорение кода приветствуется. В данном случае для ускорения применены статические таймеры. Дело тут не в том, что инкремент статического таймера производится быстрее, чем таймера задачи, а в том, что таймеры нужны только двум задачам из трех активных. И избавляясь от обработки одного неиспользуемого таймера, мы выигрываем драгоценные такты.

Переменная **s_cCurSample** описана как static, т.к. нам важно сохранение ее значения после переключения на другие задачи. Функция **CopySample** просто копирует массив из ROM-памяти в массив Sample:

```
void CopySample (char c)
{
    char n;
    c &= 3;
    for (n = 0; n < 64; n++) Sample[n] = SAMPLES[c][n];
}
```

Клавиатура

Задача чтения состояния клавиатуры каждые 10 мс опрашивает все 6 строк с кнопками. Перед первым опросом обнуляются все значения порогов для всех столбцов (аналоговых входов). Подпрограмма чтения строки проверяет эти переменные и, если они нулевые, сохраняет туда считанные с аналоговых входов значения за вычетом 15% барьера.

Примечание. *При использовании прокладки между пальцем и пластиной емкостного датчика барьер должен стоять выше.*

```
void Task_Keyboard (void)
{
    static char n;
    static char s_cChanged;

    //--------------------------------------------------------------------------
    //  Один раз выполняем чтение, чтобы точно быть уверенными в том, что все
    //  TRIS'ы аналоговых входов установлены в "0" (на выход) для разряда
    //  конденсаторов
    //--------------------------------------------------------------------------

    ReadRow(0);

    //--------------------------------------------------------------------------
    //  При первом запуске все пороги устанавливаем в 0
    //--------------------------------------------------------------------------

    for (n = 0; n < KBD_COLUMNS; n++) KBD.Porogs[n] = 0;

    for (;;)
    {
        //--------------------------------------------------------------------------
        //  Перед измерением состояния всех кнопок устанавливаем маску для первой
        //  кнопки
        //--------------------------------------------------------------------------

        KBD.cDataPos  = 0;                  // Номер байта
        KBD.cDataMask = 0x01;               // Маска бита в байте
        s_cChanged = 0;                     // Признак того, что состояние какой-то
                                            // клавиши было изменено

        for (n = 0; n < KBD_ROWS; n++)      // Цикл по всем строкам
        {
            s_cChanged |= ReadRow(n);       // Измерение всех датчиков в строке
                                            // может длиться до 500 мкс, поэтому
                                            // после прочтения каждой строки
                                            // переключаем контекст
            OS_Yield();
        }

        //--------------------------------------------------------------------------
        //  Если были изменения в кнопках, то отправляем сообщение задаче
        //  формирования звуковых переменных
        //--------------------------------------------------------------------------

        if (s_cChanged)
        {
            OS_Msg_Send_Now(msg_KBD, (OST_MSG) KBD.Data);
        }

        OS_Stimer_Delay(ST_KEYBOARD, 10 ms);
    }
}
```

Здесь так же, как и в задаче **Task_Button** для формирования задержки применен статический таймер.

Теперь рассмотрим основную функцию клавиатуры, ту, которая занимается чтением состояний емкостных датчиков.

```
char ReadRow (char row)
{
    char        m, a, i, k;             // Вспомогательные переменные
    char        col;                    // Текущий канал
    TColumn     Column;                 // Для сокращения кода обработки канала его
                                        // параметры копируются из ROM в переменную

    static char s_Changes[KBD_SIZE];    // Изменения в состояниях кнопок
                                        // для подавления дребезга
    static bit  s_bChanged;             // Переменная для возврата

    //--------------------------------------------------------------------------
```

```
*ROWS[row].pPort |= ROWS[row].cMask;// Управляющий выход для линии в "1"

s_bChanged = 0;                     // Изначально считаем, что изменений
                                    // не было

//*****************************************************************************
//   Цикл по всем каналам
//*****************************************************************************

for (col = 0; col  < KBD_COLUMNS; col++)
{
    //-------------------------------------------------------------------------
    // Копируем параметры канала в переменную
    //-------------------------------------------------------------------------

    Column.pPort       = COLUMNS[col].pPort;
    Column.pTris       = COLUMNS[col].pTris;
    Column.cMask       = COLUMNS[col].cMask;
    Column.cADCChannel = COLUMNS[col].cADCChannel;

    //-------------------------------------------------------------------------
    //  Выбираем канал ADC
    //-------------------------------------------------------------------------

    CHS0 = 0;
    CHS1 = 0;
    CHS2 = 0;
    if (Column.cADCChannel & 0x01) CHS0 = 1;
    if (Column.cADCChannel & 0x02) CHS1 = 1;
    if (Column.cADCChannel & 0x04) CHS2 = 1;

    #if defined(_16F887) || defined(_16F886) || defined(_16F690)
    CHS3 = 0;
    if (Column.cADCChannel & 0x08) CHS3 = 1;
    #endif

    //-------------------------------------------------------------------------
    //  Начинаем измерение
    //-------------------------------------------------------------------------

    GIE = 0;                        // На время заряда конденсатора блокируем
                                    // прерывания, чтобы получить фиксированную
                                    // паузу

    *Column.pTris |=  Column.cMask; // Начать заряд переводом порта на вход

    for (m = 0; m < 3; m++) NOP();  // ПАУЗА для заряда.

    GODONE = 1;                     // Начинаем АЦ-преобразование
    GIE = 1;                        // Теперь прерывания можно разрешить

    while (GODONE) continue;

    //-------------------------------------------------------------------------
    //  Измерение закончено, теперь разряжаем конденсатор, читаем результат
    //  и формируем массив нажатых кнопок
    //-------------------------------------------------------------------------

    *Column.pTris &= ~Column.cMask;
    *Column.pPort &= ~Column.cMask; // Разряжаем конденсатор переводом входа на
                                    // выход и установкой "0" на нем
    a = ADRESH;

    //-------------------------------------------------------------------------
    //  Устанавливаем порог срабатывания, если он еще не установлен
    //-------------------------------------------------------------------------


    m = KBD.Porogs[col];            // Для сокращения кода копируем элемент
                                    // массива в переменную
    i = 0;
    if (a < m) i = KBD.cDataMask;   // Сравниваем результат АЦП с порогом для
                                    // данного канала. Если результат меньше
                                    // порогового значения, значит,
                                    // емкость на входе превышала допустим
                                    // ое значение и кнопка считается нажатой

    if (!m)                         // Если значение порога еще не установлено,
    {                               // то формируем его как 88% от ADRESH
        m = a >> 3;
        KBD.Porogs[col] = a - m;
    }

    //-------------------------------------------------------------------------
    //  Установка нового значения кнопки с подавлением дребезга
    //-------------------------------------------------------------------------

    m  =  KBD.Data[KBD.cDataPos];   // Для сокращения кода работаем не с элементом
    k  =  s_Changes[KBD.cDataPos];  // массива, а с фиксированной переменной

    //-------------------------------------------------------------------------
    if ((m ^ i) & KBD.cDataMask)
    {                               // Состояние кнопки изменилось:
        if (!(k & KBD.cDataMask))   //   Только что
            k |=  KBD.cDataMask;    //     Устанавливаем признак изменения бита

        else                        //   Не только что
        {
            m ^=  KBD.cDataMask;    //     устанавливаем новое значение кнопки
            s_bChanged = 1;         //     Формируем переменную для возврата
            k &= ~KBD.cDataMask;    //     Сбрасываем признак изменения бита
        }
    //-------------------------------------------------------------------------
    } else {                        // Состояние кнопки не изменилось:
        k &= ~KBD.cDataMask;        //   Сбрасываем признак изменения бита
    }
    //-------------------------------------------------------------------------

    KBD.Data[KBD.cDataPos] = m;     // Восстанавливаем значения массивов из
    s_Changes[KBD.cDataPos] = k;    // временных переменных

    //-------------------------------------------------------------------------
    //   Устанавливаем маску для следующей кнопки
    //-------------------------------------------------------------------------

    KBD.cDataMask <<= 1;
    if (!KBD.cDataMask)
    {
        KBD.cDataMask = 0x01;
```

```
            KBD.cDataPos++;
        }

    };

    *ROWS[row].pPort &= ~ROWS[row].cMask;   // Сбросить управляющий выход линии

    return s_bChanged;
}
```

Обратим внимание на строку формирования паузы:

```
    for (m = 0; m < 3; m++) NOP();
```

В данном случае пауза рассчитана для сопротивления в RC-цепочке, равное 100K, что обеспечивает заряд емкости примерно до Vdd/2. При установке резисторов других номиналов константу в цикле желательно (но не обязательно) пересчитать пропорционально, т.е. для 200K нужно будет считать до 6.

Звук

Эта задача будет получать информацию о состоянии кнопок клавиатуры (через сообщение от **Task_Keyboard**) и формировать переменные управления звуком для «Синтезатора». При получении сообщения состояния кнопок копируются во внутренний массив для обработки, поскольку обработка предусматривает модификацию данных в этом массиве. После получения сообщения пробегаемся по всем переменным типа **TSound**, содержащим информацию о том, на каком канале какая нота воспроизводится, с целью:

- прекращения воспроизведения уже не нажатых клавиш;
- удаления из списка нажатых клавиш уже воспроизводимых на данный момент;
- вычисления количества свободных каналов.

После этого у нас есть переменная **cFreeSounds**, показывающая сколько звуковых каналов свободно, и список еще не обрабатываемых клавиш в массиве **Data**.

```
TSound      S[MAX_CHANNELS];    // Переменные для формирования звука

void Task_Sound (void)
{
    OST_MSG         msg;            // Переменная для приема сообщения
    unsigned char   Data[KBD_SIZE]; // Массив, куда будут скопированы состояния
                                    // клавиш
    unsigned char   cMask;          // Две вспомогательные переменны для побитового
    unsigned char   cPos;           // индексирования кнопок в массиве Data

    unsigned char   cFreeSounds;    // Переменная будет показывать, сколько свободных
                                    // каналов (не воспроизводящих) есть на
                                    // данный момент
    unsigned char   i, j;           // Вспомогательные переменные

    //--------------------------------------------------------------------------

    for (;;)
    {
        //----------------------------------------------------------------------
        //   Ждем изменения состояния кнопок.
        //   Копируем состояния кнопок в массив Data
        //----------------------------------------------------------------------

        OS_Msg_Wait(msg_KBD, msg);

        for (i = 0; i < KBD_SIZE; i++) Data[i] = ((char*)msg)[i];

        //----------------------------------------------------------------------
        //  Из списка нажатых кнопок удаляем те, которые в данный момент
        //  уже воспроизводятся. Одновременно считаем, сколько свободных каналов
        //  имеется на данный момент.
        //----------------------------------------------------------------------

        cFreeSounds = 0;

        for (i = 0; i < MAX_CHANNELS; i++)      // Пробегаемся по всем каналам
        {
            if (S[i].key == 0)                  // Если данный канал "молчит", то
            {                                   // увеличить счетчик свободных каналов
                cFreeSounds++;
                continue;
            }

            j = S[i].key - 1;                   // Формируем адрес бита в массиве Data,
            cMask = 1 << (j & 7);               // соответствующего текущему каналу
            cPos = j >> 3;

            if (Data[cPos] & cMask)             // Если кнопка все еще нажата, то
                Data[cPos] &= ~cMask;           // Убираем ее из списка нажатых кнопок
            else
            {
                cFreeSounds++;                  // Иначе прекращаем звук и увеличиваем
                S[i].key = 0;                   // счетчик свободных каналов.
            }
        }

        //----------------------------------------------------------------------
        //  На данный момент cFreeSound содержит число незадействованных каналов,
        //  которые можно использовать для воспроизведения звуков для вновь
        //  нажатых клавиш
        //----------------------------------------------------------------------

        cMask = 0x01;   // Поиск клавиш начинаем с первой
        cPos = 0;
        j = 0;          // Счетчик кнопок
        i = 0;          // Счетчик каналов

        while ((j < KBD_KEYS) && cFreeSounds)
        {
            if (Data[cPos] & cMask)     // Клавиша нажата?
            {                           // Да.
                while (S[i].key) i++;   // Ищем свободную ячейку

                                        // Формируем звуковую переменную:
                S[i].F = Freq[j];       // Устанавливаем частоту
                S[i].f = 0;             // Начальная фаза
                S[i].key = j + 1;       // Запоминаем номер воспроизводимой клавиши
                cFreeSounds--;          // Уменьшаем счетчик свободных каналов
            }

            j++;                        // Берем следующую кнопку для анализа
            cMask <<= 1;
```

```
            if (!cMask)
            {
                cMask = 0x01;
                cPos++;
            }
        }
    }
}
```

Синтезатор

Как мы уже решили раньше, подпрограмма синтезатора звука будет помещена внутрь прерывания по TMR2. Таймер 2 у нас настроен и для отсчета тактов модуля ШИМ, и для генерации прерываний. ШИМ у нас выбран максимально возможной частоты для 20 МГц и 8-разрядного разрешения, т.е. 78КГц. Теперь нам нужно выбрать постделитель для TMR2 такой, чтобы обеспечить частоту семплирования 20КГц. Очевидно, что ближайшим значением будет 4 (при этом мы получим частоту 19500). Итак, каждые 51.2 мкс вызывается прерывание, в котором генерируется звук, а именно - скважность импульсов ШИМ-сигнала.

```
void interrupt isr (void)
{
    static  unsigned    char    prs;        // Предделитель для вызова OS_Timer
            signed      int     temp_dac;   // Сумма мгновенного значения
                                            // сигнала для всех каналов
            unsigned    char    m_cDAC;     // Переменная для вывода через ШИМ


    TMR2IF = 0;
    temp_dac = 0;

    //--------------------------------------------------------------------------
    //  Формируем мгновенное значение суммы всех каналов
    //--------------------------------------------------------------------------

    SOUND(0);
    SOUND(1);
    SOUND(2);
    SOUND(3);
    SOUND(4);
    SOUND(5);
    SOUND(6);
    SOUND(7);

    temp_dac >>= 3;     // Т.к. 8 каналов, то сумму делим на 8.

    //--------------------------------------------------------------------------
    //  Выводим полученное значение через ШИМ
    //--------------------------------------------------------------------------

    m_cDAC  = *((char*)&temp_dac+0) + 0x80;
    m_cDAC >>= 2;
    CCP_bit1 = 0;
    CCP_bit0 = 0;
    if (temp_dac & 2) CCP_bit1 = 1;
    if (temp_dac & 1) CCP_bit0 = 1 ;
    CCPR1L = m_cDAC;

}
```

Обратим внимание на вызов макросов SOUND(x). Этот макрос написан просто для удобства добавления/удаления каналов в зависимости от требований к качеству звука и тактовой частоты контроллера (Например, понизив тактовую частоту в два раза, мы за 50 мкс будем успевать обработать только 4 канала; или, снизив частоту семплирования до 10 КГц, мы сможем обработать 16 каналов). Сам макрос выглядит так:

```
#define SOUND(x)                                        \
    if (S[x].key) {                                     \
        temp_dac += Sample[*((char*)&S[x].f+1) & 0x3F]; \
        *((char*)&S[x].f+1) += *((char*)&S[x].F+1);     \
        *((char*)&S[x].f+0) += *((char*)&S[x].F+0);     \
        if (CARRY) *((char*)&S[x].f+1) += 1;            \
    }
```

После проверки активности канала по полю **cKey** мы из массива, где хранится оцифрованный период для конкретного музыкального инструмента, в соответствии с текущей фазой сигнала (поле **f**) выбираем нужное значение и прибавляем его к общей сумме **temp_dac**. После этого сдвигаем фазу на шаг, зависящий от частоты ноты, которая проигрывается на данном канале.

Теперь в прерывание осталось добавить обработку системных таймеров. Выбираем интервал для таймера равный 10 мс, или двумстам вызовам прерывания.

```
    //--------------------------------------------------------------------------
    //  1 раз в 200 вызовов (200 * 50мкс = 10мс) вызываем системный таймер
    //--------------------------------------------------------------------------
    if (!--prs)
    {
        OS_Timer();     // Обработка системных таймеров
        prs = 200;
    }
```

main()

Здесь будут проинициализированы периферия и операционная система, а также будут созданы задачи и запущен планировщик. Все задачи имеют одинаковый высший (нулевой) приоритет.

```
void main (void)
{
    //--------------------------------------------------------------------------
    //  Инициализация периферии
    //--------------------------------------------------------------------------

    Init();

    //--------------------------------------------------------------------------
    //  Инициализация системы
    //--------------------------------------------------------------------------

    OS_Init();

    //--------------------------------------------------------------------------
    //  Создание задач (все задачи имеют одинаковый высший приоритет)
    //--------------------------------------------------------------------------

    OS_Task_Create(0, Task_Sound);
    OS_Task_Create(0, Task_Button);
    OS_Task_Create(0, Task_Keyboard);

    //--------------------------------------------------------------------------
```

```
    //  Разрешаем прерывания и запускаем планировщик
    //-----------------------------------------------------------------------

    OS_EI();
    OS_Run();
}
```

Init()

Весь текст я здесь приводить не буду (его можно посмотреть в исходных текстах, прилагаемых к статье), т.к. из-за того, что эта функция предусматривает работу на 4-х разных контроллерах (16F886, 16F887, 16F690 и 16F88), то код ее довольно громоздкий из-за наличия условных директив #ifdef…#endif.

Скажу только, что в этой функции производится инициализация:

- потов ввода/вывода;
- АЦП;
- таймеров;
- модуля ШИМ;
- прерываний.

Конфигурация OSA

Для конфигурирования работы операционной системы в нашем проекте воспользуемся утилитой OSAcfg_Tool.

1. Выбираем папку, где располагается наш проект

Для этого в самом верху окна справа нажимаем кнопку **Browse**. Там выбираем путь к файлу OSAcfg.h - путь к нашему проекту («C:\TEST\PICKIT2\PIANO»). Нажимаем OK. Если файл еще не создан, то программа спросит у Вас, действительно ли Вы хотите создать этот файл. Смело отвечаем «Yes» и идем дальше.

2. Выбираем имя проекта

В поле **Name** можно ввести имя проекта. Этот пункт необязателен, а имя вводится исключительно для наглядности, чтобы не путаться потом, какой файл от какого проекта. Мы введем в эту строку «ПИАНИНО».

3. Выбираем платформу

Также необязательный пункт. Служит только для того, чтобы пользователь при конфигурировании файла в реальном времени наблюдал предполагаемый расход оперативной памяти операционной системой. Для успокоения выберем платформу: 14-бит (PIC12, PIC16)(ht-picc). Теперь при изменении настроек мы автоматически в рамке **RAM statistic** будем видеть, сколько байтов в каком банке памяти израсходовано.

4. Конфигурируем наш проект

Учитывая, что мы решили не использовать приоритеты (т.е. все задачи сделать равноприоритетными), можно установить галочку напротив пункта **Disable priority**. Это сократит размер кода ядра операционной системы и ускорит работу планировщика.

Далее, нам обязательно нужно выбрать количество задач ОС, которые будут работать одновременно. В нашем случае - 3 (по количеству задач, создаваемых сервисом OS_Task_Create; как уже было сказано раньше, 4-я задача у нас не является задачей ОС и располагается в обработчике прерывания).

Учитывая, что сама программа использует много переменных, есть смысл все системные переменные затолкать в какой-нибудь верхний банк памяти, например **bank2** (в поле **OSA variables bank**).

Теперь нам нужно сказать системе, что нам требуются два статических таймера для работы. В поле **Static timers** устанавливаем 2 и в таблице ниже вводим имена идентификаторов статических таймеров: **ST_KEYBOARD** и **ST_BUTTON**. Кроме того, учитывая, что нам не потребуются задержки длиннее 256 системных тиков, установим тип статического таймера **char**.

И последнее: для ускорения обработки сервиса OS_Timer установим галочку напротив пункта **Use in-line OS_Timer()**.

5. Сохраняем и выходим

Жмем на кнопку **Save**, чтобы сохранить отредактированный файл конфигурации, и выходим из программы нажатием на кнопку **Exit**. Теперь, заглянув в созданный нами файл, мы увидим следующее:

```
/*****************************************************************************/
//
// This file was generated by OSAcfg_Tool utility.
// Do not modify it to prevent data loss on next editing.
//
// PROJECT NAME: ПИАНИНО
// PLATFORM:     HT-PICC 14-bit
//
/*****************************************************************************/

#ifndef _OSACFG_H
#define _OSACFG_H

//---------------------------------------------------------------------------
// SYSTEM
//---------------------------------------------------------------------------

#define OS_TASKS               3  // Number of tasks that can be active at one time
#define OS_DISABLE_PRIORITY       //

//---------------------------------------------------------------------------
// ENABLE CONSTANTS
//---------------------------------------------------------------------------

#define OS_USE_INLINE_TIMER        // Make OS_Timer service as in-line function

//---------------------------------------------------------------------------
// BANKS
//---------------------------------------------------------------------------

#define OS_BANK_OS             2  // RAM bank to allocate all system variables

//---------------------------------------------------------------------------
// TYPES
//---------------------------------------------------------------------------

#define OS_STIMER_SIZE         1  // Size of static timers (1, 2 or 4)

//---------------------------------------------------------------------------
// STIMERS
//---------------------------------------------------------------------------
```

```
#define OS_STIMERS             2  // Number of static timers

enum OSA_STIMERS_ENUM
{
    ST_KEYBOARD,          // Для формирования задержек в Task_Keyboard
    ST_BUTTON             // Для формирования задержек в Task_Button
};

#endif
```

Прошивка контроллера

Сборка проекта

Для работы с проектом нам нужно иметь установленную интегрированную среду MPLAB IDE [http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469&part=SW007002], установленный компилятор HI-TECH PICC STD [http://www.htsoft.com/microchip/products/compilers/picccompiler.php] (PRO-версия не подойдет).

Скачиваем, если еще не скачали, **файлы операционной системы OSA**, распаковываем этот архив на диск C: (должна получиться папка C:\OSA).

Распаковываем файл **piano.rar** в папку C:\TEST\PICKIT2. При этом внутри создастся папка **PIANO**. В MPLAB IDE открываем проект, в названии которого присутствует номер контроллера, который Вы собираетесь использовать: 886, 887, 690 или 88. Например, для демо-платы на базе 16F887 нам нужно открыть файл pk2_piano_887.mcp.

Примечание. *При распаковке в другую папку, отличную от C:\TEST\PICKIT2\PIANO, нужно будет через меню Project\Build options…\Project в закладке Directories в списке include-путей заменить путь к файлам проекта на тот, куда Вы распаковали файлы из архива.*

Выполняем сборку нажатием **Ctrl+F10**.

Прошивка

Здесь все просто:

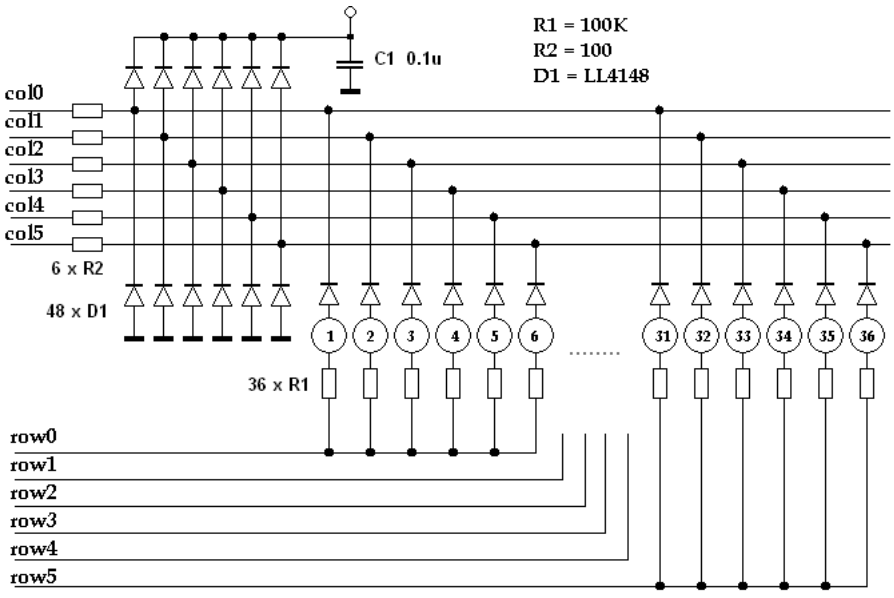
1. подключаем программатор;
2. в меню «Programmer\Select» programmer выбираем PicKit2;
3. В настройках «Programmer→Settings» выбираем <3-State on <Release from Reset>
4. запускаем программирование «Programmer\Program»;
5. освобождаем вывод MCLR «Programmer\Release from reset».

Вот и все!

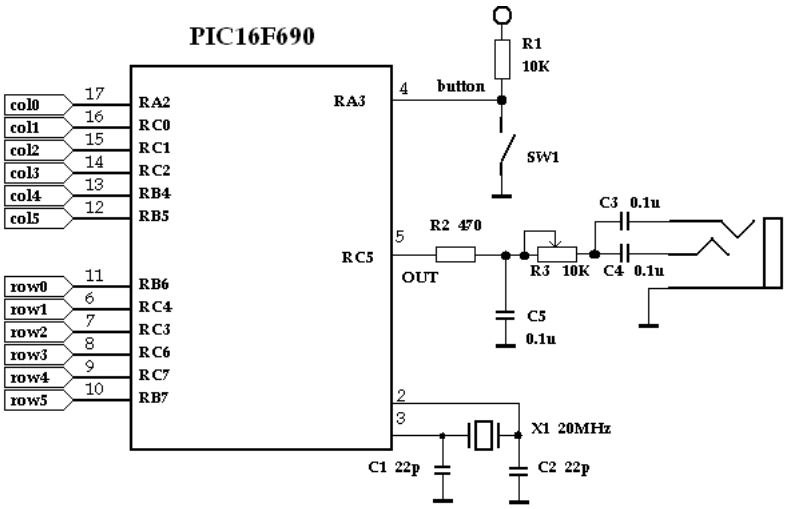
Схемы подключения клавиатурной матрицы

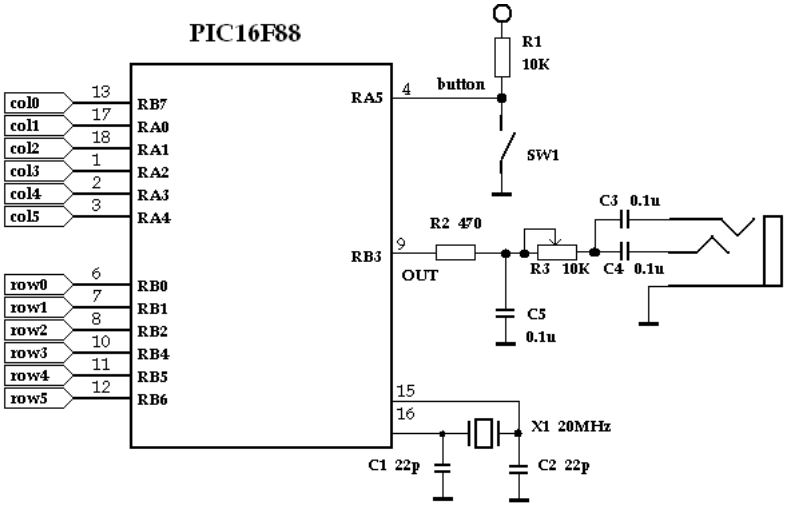
Здесь я просто приведу к каким выводам подключается матрица, чтобы работала наша программа. Рассмотрены контроллеры PIC16F88, PIC16F690, PIC16F886, PIC16F887.

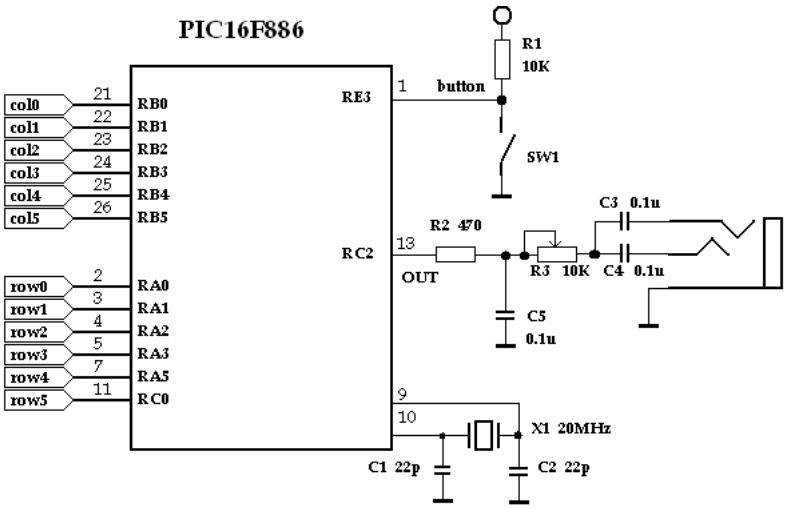
Важное примечание: *на выводах, используемых для чтения сенсорных датчиков (col1, col2 и т.д. - аналоговвые входы) не должно висеть ничего, кроме самих датчиков, т.к. любой внешний элемент будет вносить свои емкость и сопротивление.*

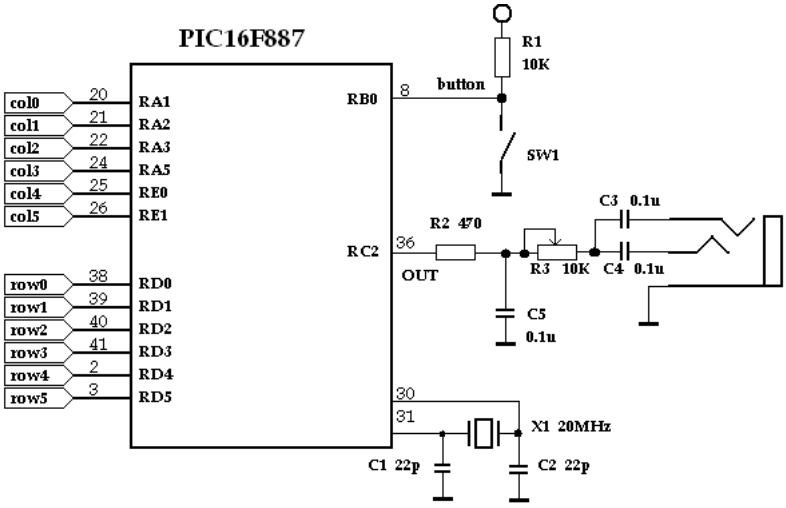


Аналоговые входы контроллера дополнительно защищены от статики согласно документа от MicrochipLayout and Physical Design Guidelines for Capacitive Sensing (PDF) [http://ww1.microchip.com/downloads/en/AppNotes/01102a.pdf]. Хоть контроллер и имеет встроенную защиту, тем не менее, во-первых, в некоторых случаях ее может оказаться недостаточно, а во-вторых, защиту имеют не все выводы (например, RA4 ее не имеет).


PIC16F690
col0
col1
col2
col3
col4
col5
row0
row1
row2
row3
row4
row5
17
16
15
14
13
12
11
6
7
8
9
10
RA2
RC0
RC1
RC2
RB4
RB5
RB6
RC4
RC3
RC6
RC7
RB7
RA3
RC5
4
5
2
3
button
R1
10K
SW1
R2 470
OUT
R3 10K
C3 0.1u
C4 0.1u
C5
0.1u
X1 20MHz
C1 22p
C2 22p


PIC16F88
col0
col1
col2
col3
col4
col5
row0
row1
row2
row3
row4
row5
13
17
18
1
2
3
6
7
8
10
11
12
RB7
RA0
RA1
RA2
RA3
RA4
RB0
RB1
RB2
RB4
RB5
RB6
RA5
RB3
4
9
15
16
button
R1
10K
SW1
R2 470
OUT
R3 10K
C3 0.1u
C4 0.1u
C5
0.1u
X1 20MHz
C1 22p
C2 22p





Заключение

Я уверен, что если у читателя хватило терпения дочитать досюда, то у него без труда хватит терпения собрать матрицу и клавиатуру и, подключив все это дело к демо-плате из набора PicKit2, насладиться игрой на собственноручно сделанном музыкальном инструменте.

Итак, в статье были рассмотрены две интересные для многих начинающих программистов темы: обработка кнопок и генерация звука. Причем рассматривались не просто кнопки, а сенсорные кнопки, что расширяет область их применения (они могут работать и в сырости, и в пыли); и мы разобрались с генерацией не просто звука, а многоканального звука. Уверен, что знания, полученные при прочтении данного пособия, а возможно и какие-то программные наработки, расширят Ваши возможности и позволят свои программы украшать интересными интерфейсными решениями.

Зачем мы воспользовались RTOS?

- Во-первых, конечно же, для того, чтобы не думать о том, как бы нам самим реализовать распараллеливание процессов.
- Во-вторых, четко разбили все на задачи, каждая из которых получилась простой наглядной. Кроме того, любая из задач может быть с легкостью модифицирована под конкретные нужды.
- Наконец, сделали наши задачи независимыми, что позволит внедрять их в другие программы, или расширить возможности этой (например, можно дописать задачу, отсылающие задаче **Task_Sound** сообщения, чтобы просто проигрывать любую зашитую в контроллер мелодию).

Наша программа написана с тем учетом, чтобы она могла быть легко модифицирована под любые пожелания:

- хотим кнопки перевесить на другие выводы контроллера - на здоровье: правим массивы **ROWS** и **COLUMNS** в файле **const.h**;
- хотим больше кнопок - пожалуйста, переопределяем константы **KBD_ROWS** и **KBD_COLUMNS** и добавляем данные о новых выводах в массивы **ROWS** и **COLUMNS**;
- хотим больше каналов - увеличиваем константу **MAX_CHANNELS** и снижаем частоту семплирования;
- хотим новых инструментов - правим массив **SAMPLES** в файле sinus.c.
- и т.д.

Все ограничивается только фантазией!

Удачи!

Ссылки

- музыкальная шкатулка от Чана (на attiny45) [http://elm-chan.org/works/mxb/report.html]
- сенсорные кнопки [http://pickit2.ru/doku.php/%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D1%8B:%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F.m
- еще про сенсорные кнопки [http://gamma.spb.ru/articles.php?i=84]

Поделки по данной статье:

- http://b612.h16.ru/pianino.htm [http://b612.h16.ru/pianino.htm]