DiscoLux

ЦВЕТОМУЗЫКАЛЬНЫЙ КОНТРОЛЛЕР

ДЛЯ ФОНТАНА

DL-Fx2

Инструкция по эксплуатации

2014

Содержание

Введение

Обращаемся ко всем читателям, интересующимся цветомузыкальными фонтанами. Идея цветомузыкального контроллера, который управлял бы работой фонтана под музыку, зародилась в голове автора задолго до того, как мы начали изготавливать и производить цветомузыкальные устройства. Однако четкого представления о том, как все должно управляться и настраиваться не существовало. Шел поиск различных путей удобной настройки контроллера, но, тем не менее, разработчику все равно приходилось лично выезжать на объект фонтана и править прошивку контроллера и логику его работы на месте. Перед нами встала задача сделать так, чтобы сам покупатель мог путем несложных манипуляций настроить контроллер «под себя». Решить данную проблему позволила идея встраивания в контроллер виртуальной машины исполняемого кода (ВМК), которая бы имела доступ ко всем программным и интерфейсным узлам контроллера.

Хочу обратить внимание, что на данный момент все, что предлагается на рынке контроллеров для цветомузыкального оборудования фонтанов не позволяет с учетом всех задержек гидравлики фонтана выдавать сигнал с опережением. Это и побудило меня создать контроллер DL-FX2.

1 Краткое описание

1.1 Назначение контроллера

Контроллер DL-FX2 предназначен для комплексного управления фонтанным оборудованием. Для этого в нем есть различные интерфейсы связи и управления, программируемая логическая связь между узлами системы, а также блок цветомузыкального сопровождения фонтана.

1.2 Основные возможности контроллера

К основным возможностям контроллера можно отнести гибкость настройки, что позволяет реализовать любую логику работы фонтана. Для этого в контроллере имеется виртуальная машина исполняемого кода (ВМК), реализующая требуемую логику управления фонтанным оборудованием.

1.2.1 Интерфейсная часть

Интерфейсная часть контроллера полностью управляется ВМК и позволяет настраивать и изменять ее параметры даже во время работы контроллера.

На борту контроллера имеются:
1 вход DMX512
1 выход DMX512
1 выход управления пикселями WS2811 WS2812 (для передачи на большие расстояния он сделан дифференциальным)
1 дискретный выход с открытым коллектором 300ма
8 аналоговых выходов с возможностью настройки из ВМК, каждый выход может перестраиваться или на выход напряжения 0...10В или на токовый выход 0...24ма
5 аналоговых входов с предельным входным напряжением 16.4В

1.2.2 Блок задержки звука

Блок задержки звука необходим для осуществления опережения управлением узлами фонтана, которые имеют запоздалую реакцию отклика от сигнала управления.

1.2.3 Виртуальная машина исполняемого кода (ВМК)

ВМК позволяет писать гибкие программы управления работой фонтана, производить чтение/запись из входных/выходных интерфейсов контроллера.

1.2.4 Блок цветомузыкального сопровождения (БЦС)

БЦС производит анализ музыкальной фонограмм с аудио входов и сохраняет результаты анализа в внутренней памяти контроллера, которая доступна из ВМК.

2. Язык программирования ВМК

Язык программирования очень сильно похож на язык программирования Си, но сильно упрощен, чтобы не вызывать больших трудностей при программировании работы фонтана.

2.1 Основные операторы языка.

Оператор – это литерал, определенные правила написания операторов образуют конструкции языка, которые заставляют ВМК выполнять некоторое действие. Операторы воздействуют на операнды (аргументы операции). Значения, над которыми оператор проводит вычисление выражения, называются операндами.
Все операторы, условно разделены на следующие категории:
- условные операторы, к которым относятся оператор условия **if** и оператор выбора **switch**;
- операторы цикла (**for**, **while**, **do while**);
- операторы перехода (**break**, **continue**, **return**) и др.

Простые операторы.
Обычные круглые скобки являются оператором, применяющимся для управления порядком вычисления выражения. Квадратные скобки применяются для индексации массива. Об операторе инкременте и декремента расскажем позже. Оператор **New** используется для создания переменных или массивов.

Математические операторы.
Язык также поддерживает основные математические операторы: умножение (*), деление (/) сложение (+), вычитание (-) и модуль (%). Назначение первых четырех операторов вам понятно; оператор модуля формирует остаток от целочисленного деления.

Операторы отношения.
К операторам отношения, называемым операторами сравнения, относятся: «меньше» (<), «меньше или равно» (<=), «больше» (>), «больше или равно» (>=), «равно» (==) и «не равно» (!=).

Логические операторы.
Довольно часто возникает необходимость проверять не одно условное выражение, а несколько. Для проверки нескольких условных выражений существуют логические операторы «И» (&&), «ИЛИ» (||) и «НЕ» (!)

2.1.1 Оператор if

If – это условный оператор, в скобках перед оператором пишется условие. После скобок точка с запятой не ставится. В фигурных скобках пишется код который выполнится, если условие истинно. Если условие ложно, код не выполнится. Else – это также условный оператор, но он выполняет свои функции, только в том случае, если условие в **if** ложно. То есть, оператор **if** можно назвать как оператор «если», а else как оператор «иначе». Давайте поставим простейшее условие. У нас есть две заранее проинициализированных переменных:

```
1  if(a < b)
2  {
3      print(“a меньше b”);
4  }
5  else
6  {
7      print(“a больше b”);
8  }
```

В данном примере выполнится кол в if, так как 5 меньше 25. Но что если a равно b. Тогда мы поступим следующим образом: (между 4-5 строкой) вставим, операторы else if и условие, в общем, так как показано на
примере ниже:

```
1  if(a < b)
2  {
3      print(“a меньше b”);
4  }
5  else if (a == b)
6  {
7      print(“a равно b”);
8  }
9  else
10 {
11     print(“a больше b”);
12 }
```

2.1.2 Оператор switch

Оператор **switch** – это оператор выбора. Он удобен в первую очередь тем, что может заменить много условий **if**, которые проверяют значение одной переменной. Представим себе такой пример. У нас есть целочисленная переменная, значение которой нужно проверить. И в зависимости от того, какое значение у данной переменной выполнить соответствующий код.

```
new var = 2;
switch(var)
{
    case 0: //аналог условия if(var == 0)
    {
        print(”Переменная var = 0”);
    }
    case 2: //аналог условия if(var == 2)
    {
        print(”Переменная var = 2”);
    }
}
```

Обратите внимание, если в фигурных скобках указано одно действие, его можно упростить. Но, что если переменная не попадает ни под одно условие в операторе switch. Тогда можно использовать **default**, он выполняется только в том случае, если switch не попадает ни в один из case.

```
new var = 2;
switch(samp)
{
    case 0: print(”Переменная var = 0”);
    case 2: print(”Переменная var = 1” );
    default: print(”Переменная var = %d” , var);
}
```

2.1.3 Оператор for

Ниже приведена конструкция цикла For:

```
for(переменная; условие; операция с переменной-счетчиком)
{
//здесь будет код, который выполнится если условие цикла истинно;
}
```

1. Цикл for имеет 4 секции.
1я секция содержит стартовый набор операторов в основном инициализируемых значений переменных используемый в цикле.
2я секция это условие выполнения следующей итерации цикла.
3я секция это действие в основном с операторами инкрементации переменных в цикле.
4я секция это код выполняемый в цикле.
2.

```
for(new a=0; a <= 3; a++)
{
    print("Hello, World!");
}
```

2.1.4 Оператор while и do{}while

Цикл **While** и любой другой цикл, повторяет свою функцию бесконечно до тех пор, пока его условие истинно. Если его условие будет ложно, цикл прекращает работу. Цикл выглядит следующим образом:

```
new a = 0;
while(a <= 3)
{
    a++;
    print("Hello, World!");
}
```

Как это работает? Все очень просто. Цикл проверяет условие, если «а» меньше или равно 5, он выполняет код заключенный в фигурных скобках. В вышеуказанном примере так оно и есть, так как наша переменная была создана, ее значение равно нулю. Инкрементируем переменную оператором ++, затем идет печать текста «Hello, World!»

Перейдем к следующему циклу, **do while**. Если **while** сначала проверял условие, а затем выполнял свою функцию, то **do while** делает все с точностью да наоборот. Сначала он выполняет код заключенный в фигурных скобках , а потом

проверяет условие.

Чтобы сделать из вышеуказанного цикла, цикл **do while**, достаточно верхнюю часть цикла вместе с условием перенести под фигурные скобки и в конце после условия после закрывающей круглой скобки поставить точку с запятой, а на то место, откуда вы вырезали **while**, поставить оператор **do**.

```
new a=0;
do
{
    a++;
    print("Hello, World!");
}
while(a <= 3);
```

2.1.4 Операторы break и continue

Данные операторы используются в циклах: **for**, **while**, **do{}while**. Оператор **break** завершает работу цикла, а оператор **continue**, пропускает оставшееся действие итерации цикла. У нас есть цикл **for** из прошлого урока:

```
new a=0;
while(a <= 3)
{
    a++;
    print("Hello, World!");
}
```

Сделаем так, чтобы в консоль сервера вывелся текст «Hello World!» не 3, а 2 раза. Вспоминаем урок про условные конструкции и создаем такое условие: если a равен 3, то завершаем работу цикла.

```
new a=0;
while(a <= 3)
{
    if(a == 2)
    {
        break;
    }
    a++;
    print("Hello, World!");
}
```

Если вместо оператора break подставить оператор continue результат получится тот же. Вы спросите, в чем же тогда разница, если результат один и тот же. В этом цикле есть один подводный камень. Если вы прочитаете последовательность действий цикла, то поймете в чем причина. А причина вот в чем: Цикл проверяет последовательно условия, начиная с 0, так как переменная a – равна 0. В первом и во втором случае, условия истинны, так как 0 и 1 меньше 3 и не равны 2. Соответственно в чат выводится 2 раза текст «Hello, World». Что break, что continue они не дают до конца выполнить цикл, то есть все функции, что находятся после самого оператора (инкремент и функцию print). Для того, чтобы увидеть отличие этих двух операторов нужно инкремент переместить перед условием, вот так:

```
new a=0;
while(a <= 3)
{
    a++;
    if(a == 2)
    {
        break;
    }
    print("Hello, World!");
}
```

В случае с оператором **break**, в консоль выведется текст «Hello, World!» всего один раз. Так как пропускается одно условие из-за инкремента, где переменная «a» должна быть равна 0. В случае с оператором **continue** текст выведется три раза, так как оператор пропускает под собой все оставшиеся функции цикла и начинает следующий цикл.

3. Встроенные(nativ) функции.

3.1 Краткие сведения

Встроенные функции в контроллер обозначаются оператором nativ перед названием самой функции, пример.

```
native DmxSend(Channels);
```

3.2 Функции Модуля Core
3.3 Функции Модуля Math
3.4 Функции Модуля Time

3.5 Функции DMX (DMX512)

Назначение:
Управление входным и выходным потоком данных протокола DMX512.

Встроенная функция: native **DmxPut**(Channel,Byte);
Записать значение **Byte** в выходной канал **Channel.**
Примечание: Функция может управлять нулевым каналом пакета. По умолчанию все устройства воспринимают номер 0го канала отличного от нуля как чужеродный пакет. Поэтому пользователь должен принудительно записать в него значение 0, если не собирается данные этого пакета использовать по своему усмотрению.

Пример:

```
New ch_motor = 2;

DmxPut(0,0);     /// Запись значения 0 в 0 нулевой
                 /// канал выхода DMX512.
DmxPut(ch_motor,12);   /// Запись значения 12 в 2 канал
                       /// выхода DMX512.
DmxPut(ch_motor+1,val);   /// Запись значения переменной val
                          /// в 3 канал выхода DMX512.
```

Встроенная функция: native **DmxSend**(Channels);
Старт отправки пакета DMX512.
Количество отправляемых каналов равно значению **Channels.**

Пример:

```
DmxSend(24);// Отправить 21 канал в выходной канал DMX512
```

Встроенная функция: native **DmxGet**(Channel);
Прочитать значение из канала **Channel** входного потока DMX512.

Пример:

```
DmxPut(0,0);

if(DmxGet(24)!=0) /// Прочитать значение из 24 канала сравнить с нулем
{   /// если значение отлично от нуля
    DmxPut(1,0);   /// Выходной 1 канал равен 0
}
else
{
    DmxPut(1,DmxGet(2)/3); /// Значение выходного 1 канал
                           /// равно значению из вх.канала ном. 2 деленному на 3
}
DmxSend(2); /// Отправить 2 канала. 0 и 1
```

3.6 Функции Модуля Ain

Назначение:
Чтение с входных аналоговых каналов.

Встроенная функция: native **AIn**(Channel);
Прочитать значение напряжения из входного аналогового канала **Channel** значение в милливольтах.
Максимальное измеряемое напряжение равно 16384 милливольт.

Пример:

```
DmxPut(0,0);

if(AIn(1)>1000)
{    /// Если на входе больше 1 вольта.
    DmxPut(1,0);   /// Выходной 1 канал равен 0
}
else
{
    DmxPut(1,DmxGet(2)/3); /// Выходной 1 канал входному каналу 2 деленному на 3
}
DmxSend(2); /// Отправить 2 канала. 0 и 1
```

3.7 Функции Модуля Aout

Установка Выходных аналоговых значений на анлоговых выходах.

Встроенная функция: native **AOutU**(Channel);
Установить значение напряжения на выходном аналоговом канала **Channel** в милливольтах.
Максимальное Выходное напряжение равно 11000 милливольт.

Пример:

```
DmxPut(0,0);

if(AIn(1)>1000)
{    /// Если на входе больше 1 вольта.
    AOutU(1,AIn(1)); /// Выходное напряжение равно входному.
}
else
{
    DmxPut(1,DmxGet(2)/3); /// Выходной 1 канал входному каналу 2 деленному на 3
}
DmxSend(2); /// Отправить 2 канала. 0 и 1
```

Встроенная функция: native **AOutI**(Channel);
Установить значение тока на выходном аналоговом канала **Channel** в микроамперах.
Максимальный выходной ток равен 24000 Микроампер(24ма).

Пример:

```
if(AIn(1)>1000)
{    /// Если на входе больше 1 вольта.
    AOutI(1,10000); /// Выходной ток 10ма зажечь светодиод
}
else
{
    AOutI(1,0); /// Потушить светодиод
}
```

3.9 Функции Модуля Fts

Настройка параметров цифровых фильтров обработки аудио фонограмм.

Параметры:

```
fts_out_val        - Выходное значение фильтра
fts_f0_val         - Центральная гармоника фильтра
fts_fr_val         - Порядок фильтра
fts_rlx_rate_up    - Скорость нарастания сигнала
fts_rlx_rate_down  - Скорость спада сигнала
fts_agc_rate_up    - Скорость нарастания усиления АРУ
fts_agc_rate_down  - Скорость спада усиления АРУ
fts_agc_db         - Глубина регулировки АРУ
fts_lga_n          - Порядок логарифмического усиления
fts_use            - Включить выключить фильтр
```

Встроенная функция: native FtsSet(Channel,Value, fts_par:par=fts_out_val);

Установить значение **Value** параметра **Par** канала фильтра **Channel.**

Пример:

```
1   FtsSet(0,15,fts_f0_val);  ///Центральная гармоника 15
2   FtsSet(0,2,fts_fr_val);   ///фильтр 2 порядка 12 Дб/Окт
3   FtsSet(0,2,fts_rlx_rate_up);
4   FtsSet(0,8,fts_rlx_rate_down);
5   FtsSet(0,14,fts_agc_rate_up);
6   FtsSet(0,8,fts_agc_rate_down);
7   FtsSet(0,32,fts_agc_db);
8   FtsSet(0,2,fts_lga_n);    /// Экспандирование 2го порядка
9   FtsSet(0,1,fts_use);      /// Задействовать фильтр
```

Встроенная функция: native FtsGet(Channel,&Value, fts_par:par=fts_out_val);
Получить значение **Value** параметра **Par** канала фильтра **Channel.**

Пример:

```
1   New Value = 0;
2   FtsGet(0,Value,fts_f0_val);    ///Центральная гармоника
3   FtsGet(0,Value,fts_fr_val);    ///
4   FtsGet(0,Value,fts_rlx_rate_up);
5   FtsGet(0,Value,fts_rlx_rate_down);
6   FtsGet(0,Value,fts_agc_rate_up);
7   FtsGet(0,Value,fts_agc_rate_down);
8   FtsGet(0,Value,fts_agc_db);
9   FtsGet(0,Value,fts_lga_n);
10  FtsGet(0,Value,fts_use);
```

3.10 Функции Модуля Adv

Модуль опережения управления оборудованием.

Встроенная функция: native **SetAdvTicks**(Channel,Ticks);
Установить опережение канала **Channel** модуля Adv в **Ticks.**

Встроенная функция: native **SetAdvNewVal**(Channel,&Value);
Задать новое входное значение **Value** для канала **Channel** модуля Adv.

Встроенная функция: native **GetAdvVal**(Channel);
Получить новое значение **Value** для канала **Channel** модуля Adv с учетом опережения.