

Пишем CSP2018 Invitro

Статья-пример SDCC-NOINIT

Оглавление

Предупреждение.....	3
Введение.....	3
1. Из чего же? Из чего же? Из чего же сделана наша программа?.....	4
1.1. Функциональные блоки программы.....	4
1.2. Как собирать? Откуда запускать?.....	6
1.3. Исходный текст — на блоки он побит, на файлы разделён.....	7
2. main() функция — начало всех процессов.....	8
3. Бежали блики по реке.....	11
4. Музыкальное сопровождение.....	12
4.1. О сколько нам мелодий чудных наш плеер быстрый проиграл!.....	12
4.2. Льется музыка-музыка-музыка, то печали, а то — веселя.....	13
4.3. Мелодий круглые монетки.....	13
4.4. Мерцают буквы нотам в такт, отрада сердца — эквалайзер.....	14
5. Свечи загадочное пламя дрожит — что видишь в нём?.....	14
6. Сменяют лики и картины друг друга плавно по чуть-чуть.....	14
7. Бежит строка и чуть дрожит — приветы в ней и пожеланья.....	14

Нет неразрешимых проблем, есть неприятные решения.

Предупреждение

Автор ни в коем случае не считает свое мнение единственно правильным и может его изменить или дополнить под действием разумных аргументов.

Также я не претендую на истину в последней инстанции, не считаю себя великим разработчиком систем как для Z80, так и систем вообще.

Посему прошу простить и понять мои ошибки, неточности и прочие места, которые вам придется не по вкусу.

Если же вам есть, что сказать — то можно всегда написать мне об этом.

Также хочется сказать большое человеческое спасибо WBRy за конструктивную критику и вообще за то, что он смог подвинуть меня на написание CSP2018 Invitro.

SfS, автор сего труда.

Введение

Написав CSP2018 Invitro я подумал, что это штука бесполезная и ненужная. И впал в уныние. Действительно, уже тысячи мне подобных, неподобных и бесподобных программистов писали на спектре миллионы всяких разных демо, где что-то летало, что-то мерцало и что-то звучало. Разумеется, как приглашение на фестиваль, эта программа заслуживает внимания. Но это внимание сведётся к одноразовому просмотру и не более.

Но, выпив квасу, закусив шашлыком и придя в благодушное настроение, я решил, что труд пропадать даром не должен.

В процессе написания CSP2018 Invitro мной были найдены немало ошибок в библиотеках SDCC-NOINIT и дописано немало полезных функций. Чтобы эти титанические усилия не пропали даром, я решил написать небольшую статью о том, как создаётся программа с помощью системы сборки проектов для ZX SDCC-NOINIT.

Итак, это статья о том, как работает CSP2018 Invitro изнутри. С разбором исходного кода и решений в нём применённых. Ну и конечно же — с примерами использования библиотечных функций.

Оговорюсь, что на деталях, описанных в документе «ZX SDCC NOINIT Система сборки проектов для Z80 на базе пакета SDCC» я останавливаться подробно не буду.

Полезные ссылки.

Документ «ZX SDCC NOINIT Система сборки проектов для Z80 на базе пакета SDCC» доступен тут: <https://github.com/salextpuru/sdcc-noinit/blob/master/doc/sdcc-noinit.pdf>.

Исходные тексты CSP2018 Invitro расположены внутри системы сборки проектов для ZX SDCC-NOINIT тут: <https://github.com/salextpuru/sdcc-noinit/tree/master/apps/csp2018invitro>.

1. Из чего же? Из чего же? Из чего же сделана наша программа?

Если ответить формально — то из каталогов и файлов. Но такой ответ вряд ли устроит кого-то. Так что опустимся до подробностей.

Согласно концепции SDCC-NOINIT каждая программа находится в отдельном подкаталоге каталога **apps/**. Поэтому все исходные коды проекта **csp2018invitro**, за исключением библиотек, находятся в каталоге **sdcc-noinit/apps/csp2018invitro/**, где **sdcc-noinit/** - корневой каталог системы сборки SDCC-NOINIT. Исходные тексты библиотек находятся в каталоге **sdcc-noinit/libsrc/**.

Будем рассматривать программу «сверху вниз». От внешнего вида, музыки и управления — к исходным кодам на С и Ассемблере.

1.1. Функциональные блоки программы

Чтобы понять дальнейшее, лучше всего запустить **csp2018invitro** на живом спектре или (для извращенцев) на эмуляторе. Для те, кто находится далеко и не захватил с собой спектрума, я привожу снимок экрана (рис.1).



Рис. 1: Экран программы **csp2018invitro**

Вся эта красота, кажущаяся единым целым, состоит из нескольких частей. И каждая часть управляется отдельным программным блоком.

Перечислим части экрана, соответствующие отдельным программным блокам (рис.2).



Рис. 2: Экран программы *csp2018invitro* (функциональные блоки)

Блок 1 (сверху в центре экрана). Рисунок цветного логотипа CSP по которому время от времени пробегает светлая полоска.

Блоки 2 (кружочки слева и справа от блока 1). Эти кружочки символизируют номер мелодии. Зелёный кружочек визуально показывает какая мелодия проигрывается в данный момент.

Блок 3 (горящая свеча слева и надпись «28-29 Июля Новосибирск (C) 2018 SfS» справа). Это на самом деле единый спрайт. Единственный движущийся объект связанный с этим спрайтом — пламя свечи.

Блок 4 (плавно сменяющиеся надписи; «Портал zx.pk.ru» на рисунке). Это отдельный программный блок, который выводит эффект плавной смены спрайтов поверх блока 3. На месте блока 4 в спрайте блока 3 специально оставлена пустота.

Блок 5 (надпись-эквалайзер CSP2018). Цвет букв меняется в зависимости от громкости и частоты музыки. Каждая буква соответствует одному параметру.

Блок 6 (бегущая строка). О блоке 6A, являющемся частью блока 6, поговорим позже.

Кроме экранных процедур есть блоки управления музыкой, блок опроса клавиатуры, блок обработки прерываний, которые происходят в начале кадра 50 раз в секунду (или каждые 20миллисекунд).

Каждый программный блок — это независимая программная единица, включающая в себя функции обработки того или иного типа данных, возможно данные и обязательно интерфейс для взаимодействия с другими программными блоками.

В идеальном случае каждый программный блок можно взять и без изменений перенести в другую программу.

1.2. Как собирать? Откуда запускать?

Заглянем в каталог **sdcc-noinit/apps/csp2018invitro/** и посмотрим, что там находится.

Концепция SDCC-NOINIT предполагает наличие для каждой программы двух файлов — файла конфигурации сборки **build.mk** и файла конфигурации программы **config.mk**.

Содержимое **config.mk**:

```
# Start address of module/program Адрес загрузки и запуска
START=0x6000

# Stack Size Размер стека
SSIZE=0x400

# Interrupt on start: 0-disable, 1-enable Прерывания запрещены
DISTART=0

# Store caller stack (1) or self stack (0) Программа имеет собственный стек
CALLERSTACK=0
```

Надеюсь, тут всё понятно. Если перевести на человеческий язык, то можно сказать, что: программа загружается с адреса **0x6000**, запускается программа с того же адреса; в момент запуска программы прерывания запрещены; программа использует собственный стек размером 1Кбайт.

Содержимое **build.mk**:

```
# App name Название программы (файлов .bin .tar и т.п.)
APP=csp2018inv

# Object files Список объектных файлов
OBJ=main.rel music.rel printscale.rel
OBJ+=cspLogo.rel spr2018.rel logos.rel uSctrollText.rel
OBJ+=musbtn.rel candle_flame.rel equalizer.rel
OBJ+=jungle_pt3.rel mus010_pt3.rel a_incom_pt3.rel kosmos_pt3.rel
mozart_stc.rel opium_pt2.rel

# LIBS (user-defined) if LIBS not defined then ALL libraies will be used.
# Список используемых библиотек
LIBS=libim2 libzxkbd libspr libay libfonts librnd libz80
```

Тут всё тоже просто. Задаётся имя программы **csp2018inv**. Именно с таким именем будут создаваться все выходные файлы, в нашем случае **csp2018inv-basic.tap**, **csp2018inv.bin**, **csp2018inv.\$C** и **csp2018inv-code.tap**. Это всё наша программа в разных форматах.

Далее список объектных файлов с расширением **.rel**. Это откомпилированные файлы, получаемые из файлов на языке C (***.c**) или ассемблера (***.s**). Сборщик сам определяет как создавать **.rel**-файлы. Например, если указан файл **music.rel**, то сборщик ищет сначала файл **music.c** и, если такой файл существует, то компилирует его. Если файл **music.c** отсутствует, то ищется файл **music.s** и ассемблирует его. Ну и, если файл **music.s** тоже отсутствует, то сборщик останавливается с сообщением об ошибке. Кому интересно подробнее — читайте описание SDCC-NOINIT.

И, наконец, задаётся список используемых библиотек. Если список не задан, то сборщик будет пытаться использовать все библиотеки, которые откомпилировал.

1.3. Исходный текст — на блоки он побит, на файлы разделён

Продолжим изучать содержимое каталога **sdcc-noinit/apps/csp2018invitro/**. Займемся исходными текстами. Чтобы было удобнее — сведём все файлы в таблицу по функциональному признаку:

№	Файл	Назначение	Программный блок
1	main.c	Основная программа	Основной цикл. Обработчик прерываний. Приоритетные задачи.
2	cspLogo.c cspLogo.h	Вывод рисунка цветного логотипа CSP по которому время от времени пробегает светлая полоска	Блок 1 (рис.2).
3	musbtn.c	Вывод кружочков, которые символизируют номер мелодии	Блоки 2 (рис.2).
4	logos.c logos.h	Вывод надписи «28-29 Июля Новосибирск (С) 2018 SfS; Плавная смена надписей справа от свечи.	Блоки 3 и 4 (рис.2).
5	candle_flame.c	Движение пламени свечи	Блок 3 (рис.2).
6	spr2018.c spr2018.h	Вывод надписи-эквалайзера CSP2018.	Блок 5 (рис.2).
7	equalizer.c equalizer.h	Смена цвета букв в зависимости от громкости и частоты музыки. (эквалайзер)	Блок 5 (рис.2).
8	printscale.c printscale.h	Печать букв увеличенной высоты для организации бегущей строки	Блок 6А (рис.2).
9	uSctrollText.c uSctrollText.h	Бегущая строка внизу экрана	Блок 6 и 6А (рис.2).
10	music.c music.h	Управление музыкальными проигрывателями	Блок воспроизведения музыки

11	a_incom_pt3.c	Музыка (данные мелодии)	
12	jungle_pt3.c	Музыка (данные мелодии)	
13	kosmos_pt3.c	Музыка (данные мелодии)	
14	mozart_stc.c	Музыка (данные мелодии)	
15	mus010_pt3.c	Музыка (данные мелодии)	
16	opium_pt2.c	Музыка (данные мелодии)	

2. **main()** функция — начало всех процессов

Как известно, программа на языке C начинается с функции **main()**. Это не совсем так, поскольку до этой функции выполняется произвольный код инициализации. Но, в нашем случае, об этом можно забыть.

Итак, что же делает функция **main()**, находящаяся в файле **main.c** в нашей программе? Функции её просты и незатейливы:

- установка собственного обработчика прерываний и второго режима прерываний (im2);
- подготовка экрана (его очистка, вывод статических изображений на экран);
- инициализация бегущей строки;
- разрешение прерываний;
- и, наконец, запуск основного бесконечного цикла.

В основном цикле по очереди выполняется два действия:

- проверка доиграла ли до конца музыка и надо ли переходить к следующей мелодии;
- смена картинки справа от свечи через определенное время.

Исходный код функции **main()** приведён ниже:

```
int main() {
    // Установки и настройки
    CLI();

    // Свой обработчик прерываний
    im2Set();
    im2SetHandler ( im2userHandler );

    // Очистка экрана
    border0();
    winSetAtr ( 0, 0, 32, 24, 0x00, 0xFF );
    winClear ( 0,0,32,8 );
    // Вывод картинок

    // CSP (цветное вверху экрана)
    logoToScreen ( 3,0 );

    // CSP2018 (Эквалайзер внизу экрана)
    spr2018_put();

    // Середина экрана
    logos_put();
}
```



```

// Инициализация бегщей строки
InitShiftText();

// Разрешаем прерывания
SEI();

// Основной цикл
while ( 1 ) {
    // Проверка - доиграла ли до конца музыка
    // и надо ли переходить к следующей мелодии
    checkMusic();

    // Проверка и смена картинка справа от свечи
    logos_check();
}

//
return 0;
}

```

А как же остальные красявости? Музыка, свеча, блики на лого? А всё это выполняется в процедуре обработки прерываний. Причем с разным приоритетом.

Разберёмся как работает обработчик прерываний **im2userHandler()**, код которого находится всё в том же файле **main.c**.

Главной особенностью обработчика прерываний является то, что в одном обработчике реализован запуск задач двух приоритетов. На самом деле, приоритетов можно задач можно организовать сколько угодно.

Если рассматривать с точки зрения приоритета выполнения, то в нашей программе три уровня приоритетов.

Уровень, самый низкий — это процедуры основного цикла функции **main()**: смена мелодий по кругу и смена картинок справа от свечи.

Следующий приоритет 1 — это визуальные эффекты (бегущая строка, пламя свечи, блик на лого, эквалайзер); опрос клавиатуры и смена мелодии по клавишам 1..6.

И, наконец, высший приоритет 2 — это проигрывание мелодии (вызов функции проигрывателя) и отсчет времени до смены картинок.

Исходный код функции **im2userHandler()** приведён ниже:

```

static void im2userHandler() {
    // Переменная, блокирующая повторный запуск фоновых процедур
    static volatile uint8_t lock;

    // Играем следующий кусочек музыки
    if ( music_im2h ) {
        music_im2h();
    }

    // Увеличиваем
    logos_int50();
}

```

```

// Проверка повторного вызова (это ж прерывания!)
if ( lock ) {
    return;
}

// Блокируем повторный вход в этот участок кода
lock=1;
// Разрешаем прерывания.
SEI();

// Сдвиг текста
CheckShiftText();

// Отрисовка эквалайзера
equalizer_draw();

// анимация пламени свечи
candle_flame_animate();

// Шаг анимации ЛОГО
logoAniStep();

// Опрос клавиатуры (в фоне, т.к. только кнопочки 1-6 важны)
zxKbdScan();
// Проверка - а может надо сменить мелодию?
keyMusic ( zxKbdInKey() );

// Разблокируем вход в этот участок кода
CLI();
lock=0;
SEI();
}

```

Лирическое отступление о стиле кода.

Обратите внимание, что функция определена как **static void im2userHandler()**, а не просто **void im2userHandler()**. Это является хорошим тоном, так как переменная или функция объявленная как **static** является видимой только в пределах одного *.c файла. Удобство очевидно — в программе может быть сколько угодно переменных или функций с одинаковыми именами, если они находятся в разных файлах. Поэтому, если переменная или функция используется только внутри того же файла, где она определена — определяйте её как **static**. Избежите конфликтов имён. Да и вам удобнее, что не надо для одинаковых по смыслу функций выдумывать разные имена. Жизнь станет легче, прекраснее и удивительнее.

Конец лирического отступления о стиле кода.

Вернемся к приоритету выполнения задач. Очевидно, что если произойдет прерывание, то основной цикл прервется и начнется выполнение обработчика прерываний **im2userHandler()**.

В начале обработчика прерываний **im2userHandler()** объявлена загадочная переменная **static volatile uint8_t lock**. Переменная эта статическая, то есть всегда существует в памяти. Разберёмся с её назначением чуть позже.

Итак, мы попали в обработчик прерываний и сразу же выполняем задачи с высшим, в нашем случае вторым, приоритетом. Сначала проверяем — не надо ли нам проиграть

очередную ноту. Если проигрыватель включен — проигрываем ноту. Затем увеличиваем счетчик времени до смены картинок справа от свечи. Всё. Задачи с высшим приоритетом кончились.

Теперь нам надо выполнить задачи с более низким, первым, приоритетом. Сложность тут в том, что если во время выполнения этих задач вдруг возникнет прерывание, то надо такую задачу прервать, и вновь выполнить задачи с высшим, вторым, приоритетом. Но, в то же время, если задача с первым приоритетом уже выполняется, то повторно начинать её выполнение не нужно, будет плохо.

Поэтому просто разрешить прерывания после выполнения задач с приоритетом 2 нельзя. Надо сначала проверить — а не выполняется ли уже задача с приоритетом 1? Именно для этой цели и служит переменная-защёлка **lock**.

После выполнения задач с высшим приоритетом 2 значение переменной-защёлки **lock** проверяется.

Если оно равно нулю, значит функции с приоритетом 1 ещё не выполняются. В этом случае в переменную **lock** записывается 1, затем разрешаются прерывания и начинают по очереди выполняться функции с приоритетом 1.

Если проверка переменной **lock** показала, что она содержит ненулевое значение, значит функции с приоритетом 1 уже выполняются. В этом случае происходит просто выход из обработчика прерывания.

После завершения выполнения всех функций с приоритетом 1 значение переменной **lock** вновь обнуляется, разрешая запуск этих функций по приходу следующего прерывания.

Вот, собственно и всё, что содержится в файле **main.c**.

3. Бежали блики по реке

Обратимся к логотипу, обозначенному цифрой «1» на рис.2. Это просто спрайт, любезно предоставленный организаторами **CSP2018**. Но чтобы он не был неподвижно-мёртвым, было решено его оживить. Жизнь ему придаёт время от времени пробегающий «блик». Этот блик — полоса по диагонали с повышенной яркостью.

В файлах **cspLogo.c** и **cspLogo.h** описаны сам спрайт и всего две процедуры работы со спрайтом-логотипом: процедура **logoToScreen()** — выводит логотип на экран и **logoAniStep()**, которая и организует вывод пробегающего блика.

Если с первой из процедур все предельно ясно, то вторая требует некоторых пояснений.

Процедура **logoAniStep()** вызывается из обработчика прерываний с приоритетом 1 каждые 50 долей секунды. Иногда, когда процессорного времени не хватает один из вызовов может быть пропущен. Это не влияет на впечатление от блика.

В файле **cspLogo.c** определён флаг **enable**, который указывает на то, выводится ли «блик» или нет. Переменная-счетчик **delay** отсчитывает время до очередной «пробежки блика» по спрайту.

Алгоритм работы процедуры прост.

Если флаг **enable** равен 0, то при каждом вызове процедуры **logoAniStep()** происходит уменьшение счетчика **delay** на единицу. Если счетчик **delay** равен нулю, то запускается очередная «пробежка блика» по спрайту (флаг **enable** устанавливается равным единице).

Если флаг **enable** равен 1, то при каждом вызове процедуры **logoAniStep()** происходит очередной «шаг блика», т. е. вывод диагональной полоски с нормальной яркостью и вывод в следующей позиции полоски с повышенной яркостью. X-координата вывода диагональной полоски определяется переменной **pos**.

Как только полоска «пробежала» весь спрайт, вычисляется задержка до очередной «пробежки блика» (от 1 до 255 пятидесятых долей секунды); позиция X устанавливается в начало (левый нижний угол спрайта); флаг **enable** устанавливается равным нулю, запрещая «пробежку блика».

Основу манипуляции с атрибутами составляет библиотечная функция **void winSetAtr(uint8_t x, uint8_t y, uint8_t w, uint8_t h, uint8_t attr, uint8_t mask)**.

Это универсальная функция манипуляции с атрибутами. Параметры **x,y,w,h** задают соответственно координаты и размеры окна атрибутов на экране. Параметр **attr** задаёт атрибут, а параметр **mask** — битовую маску атрибута. Если нам надо поменять, например, только **INK**, то **mask=7 (007, 0x07)**, если **PAPER** — то **mask=56 (070, 0x38)**. Для **BRIGHT** **mask=64 (0100, 0x40)**, а для **FLASH** **mask=128 (0200, 0x80)**.

Разумеется, маски можно как угодно комбинировать. Например для **BRIGHT** и **INK** маска будет **mask=71 (0107, 0x47)**. Принцип понятен.

В процедуре **logoAniStep()** функция **winSetAtr()** используется далёким от оптимального способом. Но так как быстроедействие хватает, то и оптимизация не нужна.

Если вам после моих пояснений что-то не понятно — глядите в код. Там всё написано:)

4. Музыкальное сопровождение

Музыка. Как только она появляется в программе — сразу становится теплее на душе. Сколько копий было сломано о то, как её лучше играть, какое железо использовать и ~~у кого мелодичнее звенят колушки на морозе~~ кто более музыкальный музыкант.

Сразу оговорюсь — я не музыкант вообще. Медведь плясал на моих ушах и голосовых связках. Потому все вопросы о достоинствах и недостатках мелодий — к их авторам. Я лишь воспроизвожу то, что создано другими. Потому сосредоточимся на коде, который позволяет нам слушать музыку.

В нашей незатейливой программе музыка не просто играет. Во-первых, мелодий несколько и играют они по кругу одна за другой. Во-вторых, пользователь может прямо указать какую из шести мелодий он хочет услышать. В третьих, необходимо отображать на экране какая именно из мелодий в данный момент проигрывается. Кроме того, в четвёртых, на экране есть ещё и «эквалайзер».

4.1. О сколько нам мелодий чудных наш плеер быстрый проиграл!

Итак основа всего — несколько музыкальных проигрывателей. В нашем случае их два: один для проигрывания музыки в форматах **PT2/PT3** и второй для **STC**-мелодий.

Все музыкальные штучки расположены в библиотеке **libay**. В программе имеется три файла с процедурами, имеющими отношения к музыке — это **music.c**, **musbtn.c** и **equalizer.c**.

Файлы с мелодиями содержат данные и на них мы останавливаться не будем.

Файл **music.c** содержит описание всех мелодий и процедуры смены проигрываемых мелодий.

Каждый из музыкальных плееров имеет три стандартных процедуры: инициализация проигрывания мелодии; шаг проигрывания мелодии; проверка на окончание проигрывание мелодии.

Алгоритм проигрывания мелодий крайне прост:

- инициализация очередной мелодии (вызов процедуры инициализации плеера с указанием на мелодию);
- затем каждое прерывание 1/50сек вызывается процедура проигрывание шага мелодии;
- если процедура проверки на окончание мелодии скажет: «мелодия проиграна!», то инициализация следующей мелодии.

И так — по кругу без конца.

Особенность нашей программы состоит в том, что используются мелодии в разных форматах. Чтобы не городить бог весть что, для каждой мелодии имеется запись (структура типа **sMusic**), которая хранит ссылку на данные мелодии, флаги, специфичные для плеера данного формата и код формата.

Код формата нужен для того, чтобы знать — какой плеер использовать для данной мелодии.

Все записи сведены в массив **musics[]**, хранящий все описатели мелодий.

Макрос **N_MUSICS** — это просто количество мелодий, которое вычисляется при компиляции программы автоматически. Так следует поступать всегда, когда не знаете будет у вас 2 записи в массиве или 20. Пусть рутину делает компьютер. И вы не ошибетесь.

Процедура **initNewMusic()** получает на вход ссылку на описатель мелодии (структуру **sMusic**) и вызывает функцию инициализации соответствующего плеера. Кроме того, процедура **initNewMusic()** выставляет обработчик прерывания **music_im2h()** для соответствующего плеера. Именно этот обработчик вызывается для проигрывания очередного кусочка мелодии каждое прерывание с наивысшим приоритетом.

Функция **checkEndOfMusic()** получает на вход ссылку на описатель мелодии (структуру **sMusic**) и вызывает функцию проверки на окончание мелодии соответствующего плеера.

Процедура **checkMusic()** примитивна до ужаса.

В ней проверяется — закончена ли очередная мелодия с помощью вызова функции **checkEndOfMusic()**. Если мелодия закончена, то происходит переход к следующей мелодии. Номер проигрываемой мелодии хранится в переменной **musicNumber**.

Так как вызов процедуры **checkMusic()** не имеет жесткой привязки ко времени — то она вызывается в фоне с низшим приоритетом.

4.2. Льется музыка-музыка-музыка, то печалю, а то — веселя...

Как уже говорилось, пользователь может прямо указать какую из шести мелодий он хочет услышать. Для этого создана процедура **keyMusic()**. Она очень проста. На вход передаётся код клавиши, который считывается библиотечной функцией **zxKbdInKey()**. Если клавиша не нажата (код равен нулю), то ничего не происходит. Если нажаты клавиши от «1» до «6» — происходит инициализация указанной мелодии.

4.3. Мелодий круглые монетки

Шесть круглых «монеток» (область 2, согласно на рис.2.) показывают какая из мелодий проигрывается в настоящий момент.

Если вы обратили внимание, что я ни слова не сказал о процедуре **musBtnDraw()**, которая вызывается при инициализации новой мелодии в процедурах **checkMusic()** и **keyMusic()**. Именно эта процедура, описанная в файле **musbtn.c**, выводит на экран «монетки» мелодий. Все «монетки» выводятся синим цветом и только одна из них, соответствующая проигрываемой мелодии — зелёным.

Координаты «монеток» сведены в массив структур **pos[]**, описанный в том же файле **musbtn.c**. Это позволяет располагать «монетки» в произвольных координатах и легко и быстро менять их расположение при необходимости.

Процедура **spr0_out0_attr()** выводит спрайт с атрибутами. Так выводятся все «неактивные монетки».

Процедура **spr0_out0()** выводит спрайт без атрибутов, а уже знакомая нам процедура **winSetAttr()** устанавливает атрибуты в нужные значения. Так выводится «монетка», указывающая на проигрываемую мелодию.

4.4. Мерцают буквы нотам в такт, отрада сердца — эквалайзер

Эквалайзер. Красиво, загадочно и сразу придаёт программе что-то такое, чего раньше не было. Штука крайне примитивная для музыкального сопроцессора АУ.

Засунув нос в файл **equalizer.c**, видим, что вся реализация этого действия занимает жалкие несколько строк.

Надпись **CSP2018** (область 5, согласно на рис.2.) выводится один раз при инициализации экрана в процедуре **main()**. Сам спрайт и его вывод описаны в файле **spr2018.c**. Дальнейшие манипуляции с атрибутами производятся процедурой **equalizer_draw()** из файла **equalizer.c**.

Так как буквы на спрайте имеют разные размеры, то не мудрствуя лукаво, все координаты и размеры буковок сведены в массив структур **boxText[]**, что в файле **equalizer.c**.

Значения всех регистров музыкального сопроцессора АУ считываются библиотечной функцией **ayDumpGet()**. Данная функция считывает значения регистров в массив байт **ayDump[]**.

Затем, для каждой из буковок, в соответствии со значением регистра музыкального сопроцессора выставляется атрибут согласно таблице **colorTable[]**.

Процедура **equalizer_draw()** вызывается из обработчика прерывания с приоритетом 1. То есть выше, чем фоновый цикл, но ниже, чем музыкальный плеер.

Вот, собственно и всё об «эквалайзере».

5. Свечи загадочное пламя дрожит — что видишь в нём?

Свеча является частью изображения области 3, согласно на рис.2. Но пламя свечи должно двигаться. Движение сделано примитивно — 5 спрайтов, поочерёдно сменяющих друг друга.

Процедура смены спрайтов пламени свечи описана в файле **candle_flame.c** и называется эта процедура **candle_flame_animate()**. Она настолько проста, что и описывать её стыдно. Но надо — значит надо.

Счетчик **candle_flame_delay** отсчитывает 5 прерываний (одну десятую секунды) после чего происходит смена спрайта на следующий.

Счетчик **candle_flame_counter** хранит текущий номер спрайта пламени свечи.

Процедура **candle_flame_animate()** вызывается из обработчика прерывания с приоритетом 1, т.е. с тем же приоритетом, что и отрисовка эквалайзера.

Вот, собственно и всё о «пламени свечи».

6. Сменяют лики и картины друг друга плавно по чуть-чуть

7. Бежит строка и чуть дрожит — приветы в ней и пожеланья

ПРОДОЛЖЕНИЕ СЛЕДУЕТ!