Parallax Propeller.
Заметки начинающего пропеллиста. Часть 2.

igorkov / 2009-10-30 / fsp@igorkov.org

В предыдущей части мы кратко рассмотрели возможности Parallax Propeller и даже написали пару простых приложений. Теперь хочу подробнее описать архитектуру ядра Propeller и среду разработки. Если Вы хорошо владеете английским, то лучше изучить официальную документацию, она намного лучше и полнее описывает архитектуру Propeller-а.

Подробнее о «нутре» Propeller-а

Как уже было сказано ранее, Propeller содержит 8 абсолютно симметричных ядер. Классическое изображение[1] «нутра» представлено на рисунке 1.

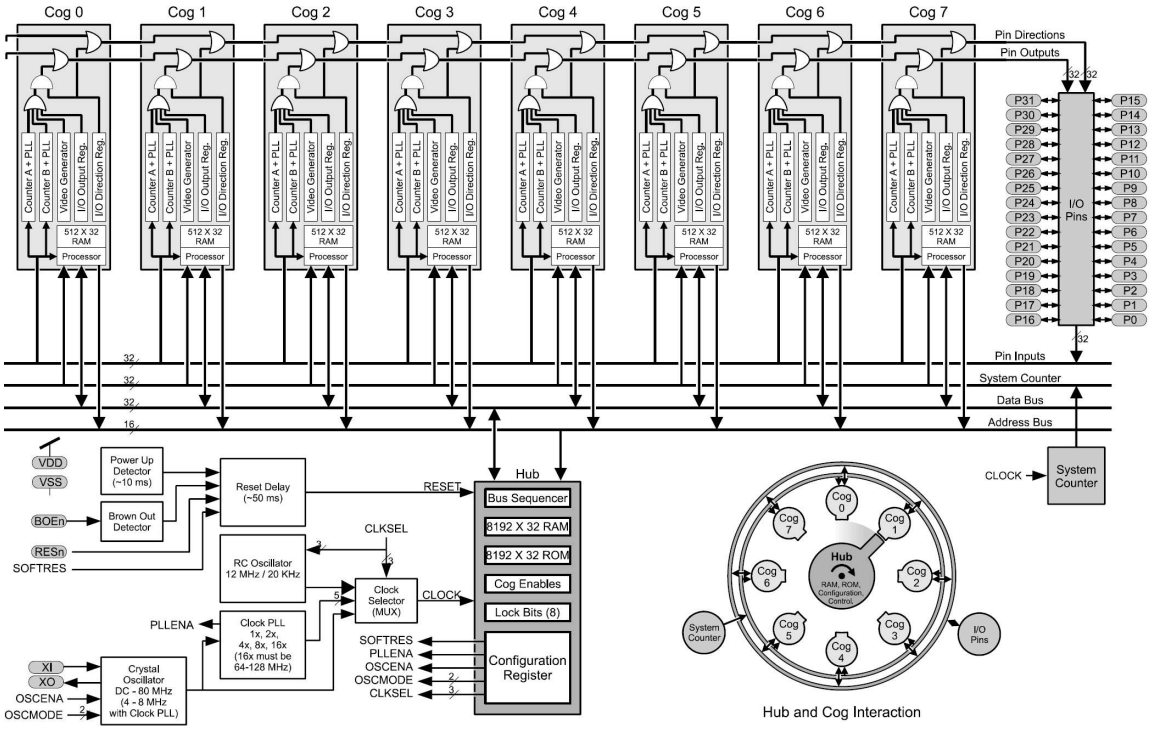

Рис. 1 – Внутреннее устройство.

Для начала, что из себя представляет каждое ядро? Это 32 разрядный вычислительный модуль. Данный модуль имеет свою локальную память, доступную только ему и свою копию регистров ввода вывода. Банк локальной памяти имеет размер 512 машинных слов, т.е. 2048 байт. В данной памяти размещается исполняемая ядром микропрограмма, в нее отображаются 16 регистров ввода-вывода, в ней должны находиться все локальные данные с которыми производятся действия. Как таковой блок регистров в ядре отсутствует, действия производятся прямиком с ячейками памяти.

У каждого ядра так же имеется небольшой набор периферии: доступ к портам ввода-вывода и 2 одинаковых таймера-счетчика. Про логику работы пинов ввода-вывода повторюсь:

1. Пин является выходом, когда хотя бы одно ядро установило его в качестве выхода.

[1] Украденное мною из офф. документации.

2. Пин находится в низком состоянии, когда хотя бы одно ядро, установившее его на выход, установило его в низкое состояние.

Эта логика хорошо отражена на рис. 1. Так же доступ к портам имеется у таймеров, именно они могут генерировать . При этом каждый (!) таймер снабжен персональным PLL и частота его работы может достигать 128МГц. А про режимы работы таймеров, думаю, придется писать отдельную статью (тем более они довольно скудно описаны в официальной документации).

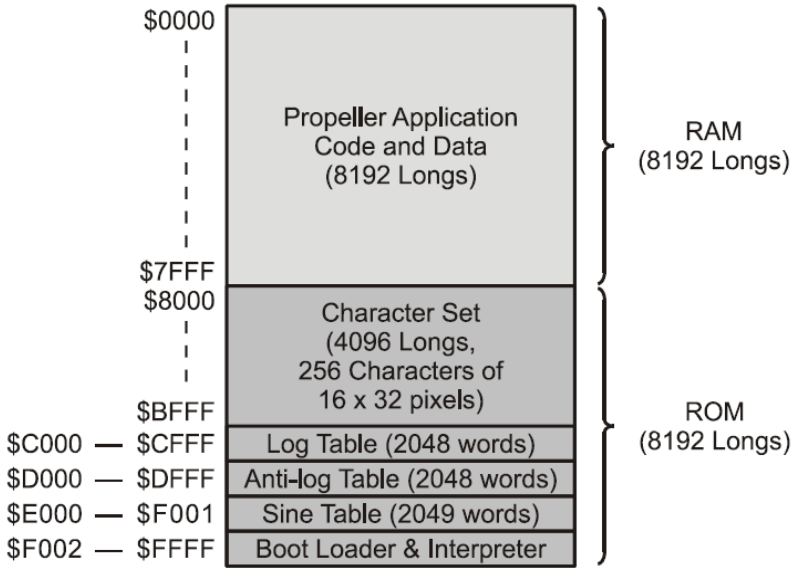

Рис. 2 – Распределение общей памяти в Propeller

Так же есть большой блок общей памяти размером 64кБ. Он разбит на 2 равные части: ПЗУ и ОЗУ. В ПЗУ лежит интерпретатор spin, математический таблицы и таблица знакогенератора. Причем ПЗУ по документации именно ПЗУ, а не какая-то разновидность ППЗУ. Структура памяти представлена на рис. 2.

Управляет доступом к общей памяти специальное устройство под названием HUB. Оно последовательно коммутирует каждое ядро к общим ресурсам, спрашивает, нужно ли ядру . Именно по этой причине, инструкции работабщие с общими ресурсами непредсказуемы по времени исполнения и могут занимать от 7 тактов (в наилучшем случае, когда HUB «на подходе») до 22 тактов в худшем случае (когда HUB только проскочил данное ядро). Именно из-за HUB-а, а именно его условного обозначения, представленном на рис. 1, Parallax Propeller скорее всего и был назван пропеллером.

И чуть не забыл упомянуть о важной особенность, уже указанная в прошлый раз: полное отсутствие системы прерываний.

Полную систему команд приводить не буду, приведу лишь некоторые команды, позволяющие понять общие принципы построения ядра. Приведенные команды я разбил на следующие подмножества:

1. Инструкции работы с общей памятью.
2. Арифметические инструкции.
3. Инструкции вызова подпрограмм.
4. Инструкции ожидания события.
5. Иструкции управления ядрами.
6. Инструкции работы с аппаратными мьютексами.

Стоит заметить, что КАЖДАЯ инстукция имеет возможность условного исполнения, не обязательно в простых случаях if/else городить конструкции из прыжков. Так же в инструкциях есть отдельные биты, отвечающие за модификацию

флагов процессора. То есть можно задать, будет ли инструкция модифицировать тот или иной флаг. Аналогично, присутвует флаг сохранения результата, позволяющий «вхолостую» выполнить почти любую иструкцию, чтобы превратить ее в аналог CMP.

Но так же стоит заметить: система команд довольно неэффективна в плане размеров получаемого кода. Особенно из-за отсутвия возможностей кодирования в командах значений и прямой адресацией всего блока локальной памяти.

Теперь о подмножествах отдельно.

Инструкции работы с общей памятью

Представлены набором инструкций загрузки/сохранения данных. Имеется 3 пары инструкций, для работы с байтовыми, вордовыми и двордовыми данными соответственно. В таблице ниже представлен двордоый вариант:

Instruction	Description	Z Result	C Result	R	Clocks
WRLONG D,S	Write D to main memory long S[15..2]	-	-	0	7..22
RDLONG D,S	Read main memory long S[15..2] into D	Result = 0	-	1	7..22

Инструкции загрузки/сохранения оперируют с адресацией через регистр, т.е. адрес (или хотя бы часть адреса) не кодируется внутри инструкции.

Арифметические инструкции

Представлены инструкциями передачи и обработки данных внутри локальной памяти ядра. Это всевозможные логические инструкции и операции арифметического сложения и вычитания. К сожалению, арифметическое умножение отмечено как future и в настоящий момент отсутвтвует.

Instruction	Description	Z Result	C Result	R	Clocks
MOV D,S	Set D to S	Result = 0	S[31]	1	4
AND D,S	AND S into D	Result = 0	Parity of Result	1	4
OR D,S	OR S into D	Result = 0	Parity of Result	1	4
XOR D,S	XOR S into D	Result = 0	Parity of Result	1	4
ADD D,S	Add S into D	D + S = 0	Unsigned Carry	1	4
SUB D,S	Subtract S from D	D - S = 0	Unsigned Borrow	1	4
CMP D,S	Compare D to S	D = S	Unsigned (D < S)	0	4

Из инструкций обработки данных сразу идет интересное следствие: работа при обработке происходит прямиком с ячейками памяти. Т.е. фактически весь блок локальной памяти представлен банком регистров, к каждому из которых имеется непосредственный доступ. Довольно удобно.

С другой стороны это и недостаток: команды довольно примитивны в плане дополнительных фишек, при этом занимают значительный объем. Подход «Load And Store» (загрузил, обработал, сохранил) все-таки эффективнее в плане объема кода.

Инструкции вызова подпрограмм и управлением потоком исполнения

На первый взгляд стандартны: вызов (команда CALL) и возврат (команда RET). Интересные вещи начинаются, когда разбираешься с данными командами, а точнее с командой RET. До сих пор не было упоминаний про регистры стеков или специализированные регистровые ячейки. Почему? Потому что их и нет!

Instruction	Description	Z Result	C Result	R	Clocks
JMPRET D,S	Insert PC+1 into D[8..0] and set PC to S[8..0]	Result = 0	-	1	4
JMP S	Set PC to S[8..0]	Result = 0	-	0	4
CALL #S	Like JMPRET, but assembler handles details	Result = 0	-	1	4
RET	Like JMP, but assembler handles details	Result = 0	-	0	4

В таблице сказано в описании RET-а: аналогично JMP. Интересно, не правда ли? Все оказывается довольно просто: При вызове функции в CALL-е ассемблером кодируется 2 адреса: куда совершаем прыжок и куда кладел адрес возврата. Прыжок – это собственно адрес функции, а возврат – адрес инструкции RET. В нее и записывается нужный адрес.

Довольно кривая система, которая накладывает следующие ограничения:

1. У функции может быть только одна точка выхода.
2. Нельзя организовать рекурсию.

А в остальном все нормально. В Spin эти ограничения обходятся программно, а на ассемблере, судя по всему предполагается написание относительно простых и компактных модулей (особенно учитывая размер локальной памяти ядра), так что особых проблем эти ограничения доставлять не должны (ну а та же рекурсия вообще считается в программировании вредной.

Инструкции ожидания события

Наверное, самый необычное подмножество в инструкциях Propeller. Из-за отсутвия системы прерываний, становится необходимым самостоятельно ловить какие-либо события. Здесь на помощь и приходят данные инструкции.

Instruction	Description	Z Result	C Result	R	Clocks
WAITPEQ D,S	Wait for pins equal - (INA & S) = D	-	-	1	5+
WAITPNE D,S	Wait for pins not equal - (INA & S) != D	-	-	0	5+
WAITCNT D,S	Wait for CNT = D, then add S into D	-	Unsigned Carry	1	5+
WAITVID D,S	Wait for video peripheral to grab D and S	-	-	0	5+

Конечно, они не были обязательными, однако без них приглось бы организовывать. А так все получается просто и понятно, без лишних команд и условий. Это позволяет уменьшить задержки и, соотвтетвенно, уменьшить время отклика.

Тем более код, зависящий от внешних событий так же становится предсказуемым по количеству тактов после момента возникновения события (т.е. нет разницы когда возникнет событие, если мы работаем в цикле с условием, то после возникновеняи события может пройти различное количество тактов до момента выхода их цикла, здесь же оно строго фиксированное).

Конкретно, можно ожидать следующих событий: изменение состояния на пинах по маске, совпадения глобального счетчика и события от таймера в режиме генерации видеосигнала.

Иструкции управления ядрами

Представлены в таблице:

Instruction	Description	Z Result	C Result	R	Clocks
COGID D	Get this cog number (0..7) into D	ID = 0	0	1	7..22
COGINIT D	Initialize a cog according to D	ID = 0	No cog free	0	7..22
COGSTOP D	Stop cog number D[2..0]	Stopped ID = 0	No cog free	0	7..22

Лично мне из описания не особо понятно каким образом работает запуск нового потока. Требуется загрузить требуемую микропрограмму в локальную память ядра и передать ей управление. Инструкция совершенно не отражает данной особенности. Дополнительной документации я тоже пока не встретил на этот счет.

Инструкции работы с аппаратными мьютексами

Меня данные инструкции немного разочаровали. Я ожидал встретить инструкцию ожидание освобождения среди них, наподобии инструкции семейства WAIT. Однако, среди них только инструкции изменения состояния и уравления. Контроль того, получилось ли захватить мьютекс придется осуществлять вручную.

Интересно, что выбор мьютексов осуществляется аппаратно из набора. Инструкция не содержит явного упоминания, что требуется использовать мьютекс под каким-то номером. Это с одной стороны немного усложняет управление (требуется передать между ядрами какой именно используется объект синхронизации), с другой стороны позволяет легко комбинировать модули, не заботясь о перектывании ресурсок, а в будущем нарастить число аппаратных мьютексов без каких-либо изменений для уже существующих микропрограмм.

Аналогичное утверждение справедливо и для инструкций работы с ядрами.

Instruction	Description	Z Result	C Result	R	Clocks
LOCKNEW D	Checkout a new LOCK number (0..7) into D	ID = 0	No lock free	1	7..22
LOCKRET D	Return lock number D[2..0]	ID = 0	No lock free	0	7..22
LOCKSET D	Set lock number D[2..0]	ID = 0	Prior lock state	0	7..22
LOCKCLR D	Clear lock number D[2..0]	ID = 0	Prior lock state	0	7..22

И напоследок, на рис. 3 приведу распиновку чипа в DIP-корпусе. Стоит заметить, она очень простая. Нет нагромождений всяких интерфейсов и привязки по пинам. Все пины расположены последовательно и не требуется запоминать или каждый раз смотреть где что расположено. А гибкие возможности назначения функций пинам за счет программной реализации большинства интерфейсов и периферии (об этом далее) позволяют очень хорошо упростить разводку платы.

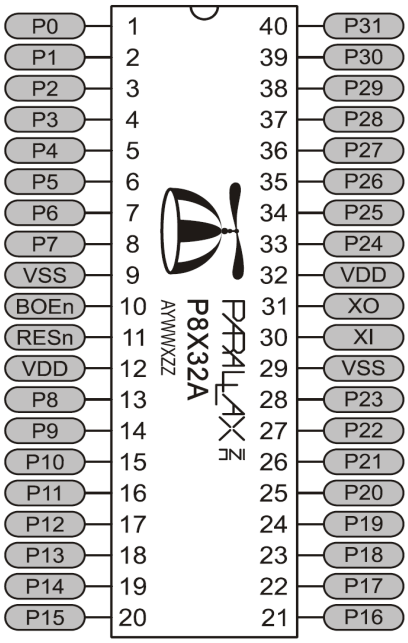

Рис. 3 – Распиновка микроконтроллера в корпусе DIP-40

Среда Propeller Tool

Общий вид среды представлен на рис. 4. Внешне она довольно простая. Меню, никаких быстрых кнопок, некоторые неудобства типа неработы сочетания «Alt+F4» и полного непонимания «великого и могучего».

Если рассматривать элементы, то это:

7. Столбец слева, верхний: здесь появляются активные файлы. Поведение окна мне не совсем понятно ☺.
8. Ниже: «проводник» с директориями и ниспадающий список с последними папками и папками стандартных библиотек (которые и открыты).
9. Содержимое открытой папки. В данном случае стандартные модули.
10. Справа большая область с кодом. Как видно, подсветка специфичная, различными цветами подсвечиваются блоки кода (константы, подключение внешних модулей и функции. Так же отдельным цветом выделяются комментарии.

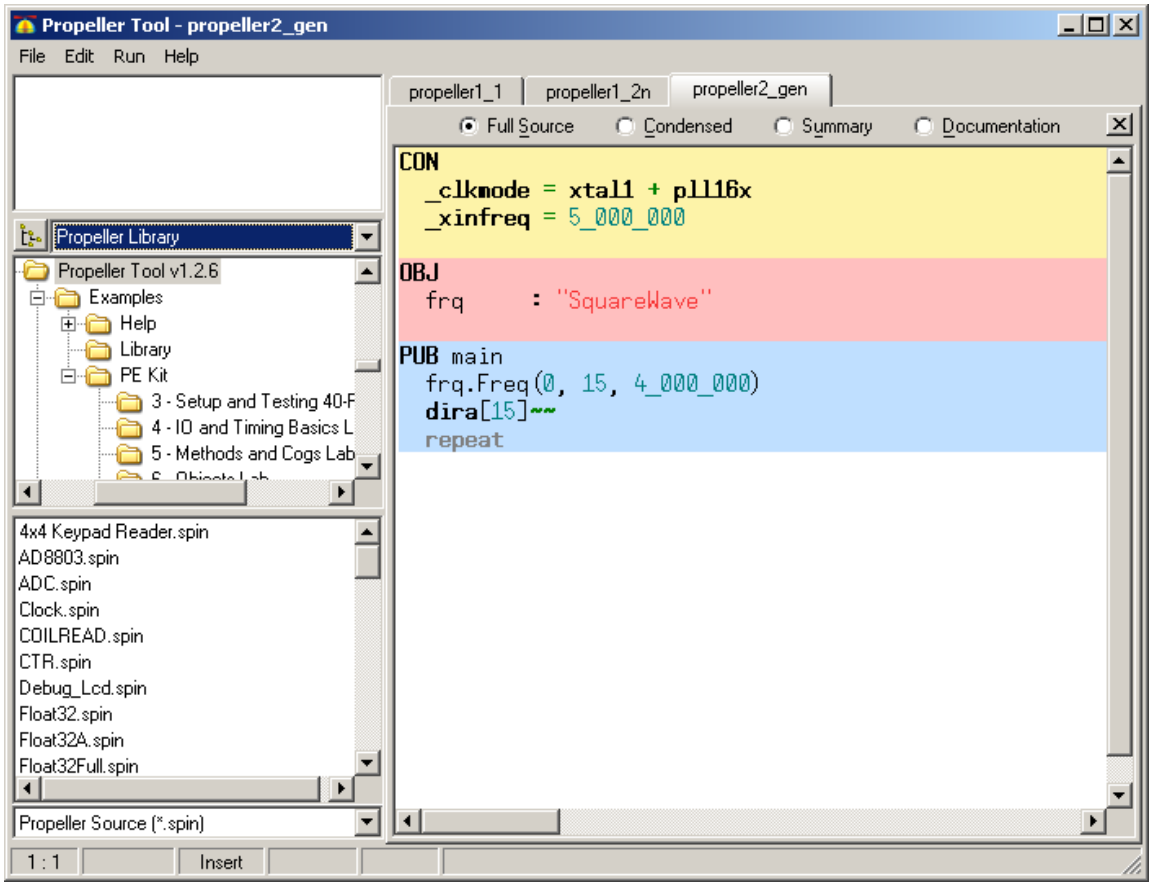

Рис. 4 – Среда Propeller Tool.

Быстрая разработка

При написании простеньких микропрограмм для пропеллера (в основном для различных тестов), сразу замечается очень интересная особенность: модульность построения программы. Весь функционал не кладется в один единственный файл, а разбивается на набор модулей, каждый из которых выполняет определенную функцию. Мало того, есть большое число таких готовых модулей.

Генерация сигналов

Чтобы наглядно продемонстрировать, чем упрощают жизнь готовые модули рассмотрим простой пример. Требуется сгенерировать меандр с какой-то определенной частотой. Можно, конечно, прочитать докуменатцию, понять какой требуется выбрать режим у таймера настроить все вручную. А можно немного побездельничать, полазить по библиотекам и примераи м обнаружить модуль «SquareWave.spin», который содержит функцию Freq. Она, как сказано в описании способна генерировать сигнал заданной частоты. Вот полученный код:

```
01| CON
02|   _clkmode = xtal1 + pll16x
03|   _xinfreq = 5_000_000
04|
05| OBJ
06|   frq     : "SquareWave"
07|
08| PUB main
09|   frq.Freq(0, 4, 4_000_000)
10|   dira[4]~~
```

```
11|    repeat
```

Из описания функции Freq, следует, что она способна генерировать сигнал с частотой от 1Гц до 128МГц ровно, с шагом в 1Гц. Так же устанавливается номер порта ввода-вывода. В примере сгенерирована частота 4МГц. Если верить осциллографу, частота устанавливается очень точно.

Теперь разберем пару ключевых моментов:

1. На строке 5-6 импортируется внешний модуль. В данном случае SquareWave (в библиотеках пропеллера можно так же найти модуль Synth, выполняющий, судя по всему, туже самую функцию). Модуль импортируется с именем frq, именно с этим префиксом становятся доступны функции из него.
2. На стр. 9 инициализируется генерация: вызывается внешняя функция Freq из модуля SquareWave.
3. Затем соответствующий пин инициируется на выход (чтобы активировать генерацию сигнала на пине) и мы зависам в бесконечном цикле.

Генерация видеосигнала

Следующий пример использования библиотек: генерация видеосигнала через VGA. Из аппаратных изменений требуется подключить VGA-разъем (рис. 5).

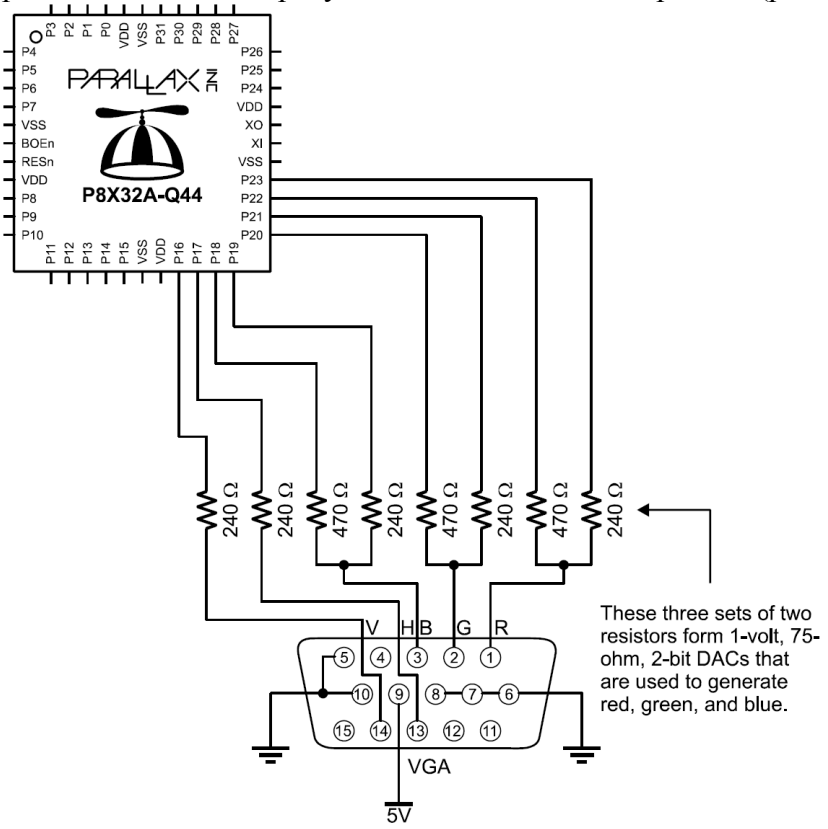

Рис. 5 – Подключение VGA-разъема к Propeller

Для генерации видеосигнала существуют библиотеки как низкого: vga, vga512x384_Bitmap, так и выского уровня: vga_text и vga_hires_text, обеспечивающие вывод текстовой информации. Рассмотрим два текстовых модуля: один работает в невысоком разрешении и потребляет немного ресурсов, а второй наоборот в высоком (если, не ошибаюсь 1024*768) и для работы уже требует намного больше ресурсов. Однако, есть и другие модули, генерящие VGA сигнал вплодь до 1600*1200, но съедающие большую часть ресурсов микроконтроллера, поэтому я их не рассматриваю.

Для примера, посмотрим что позволяет модуль vga_text:

```
01| CON
02|   _clkmode = xtal1 + pll16x
03|   _xinfreq = 5_000_000
04|
05| OBJ
06|   text : "vga_text"
07|
08| PUB start | i
09|   text.start(16)
10|   i := 0
11|   repeat
12|     text.out($C) ' set color
13|     text.out(i)
14|     text.str(string("Test... "))
15|     i++
16|     if i > 32
17|       i := 0
18|     waitcnt(CNT + 80_000_000 / 3)
```

Пример выводит на экран надпись «Test... » разными цветами 3 раза в секунду. Это обеспечивается функционалом vga_text, позволяющим задавать цвета (а так же позиции вывода на экране, очищать экран).

Немного прокомментирую код:

1. Секция CON (строки 1-3) не требует пояснений, такое уже встречалось ранее.
2. Секция OBJ (строки 5-6) содержит подключение модуля vga_text. Его функции становятся доступны с префиксом text.
3. Функция старта.

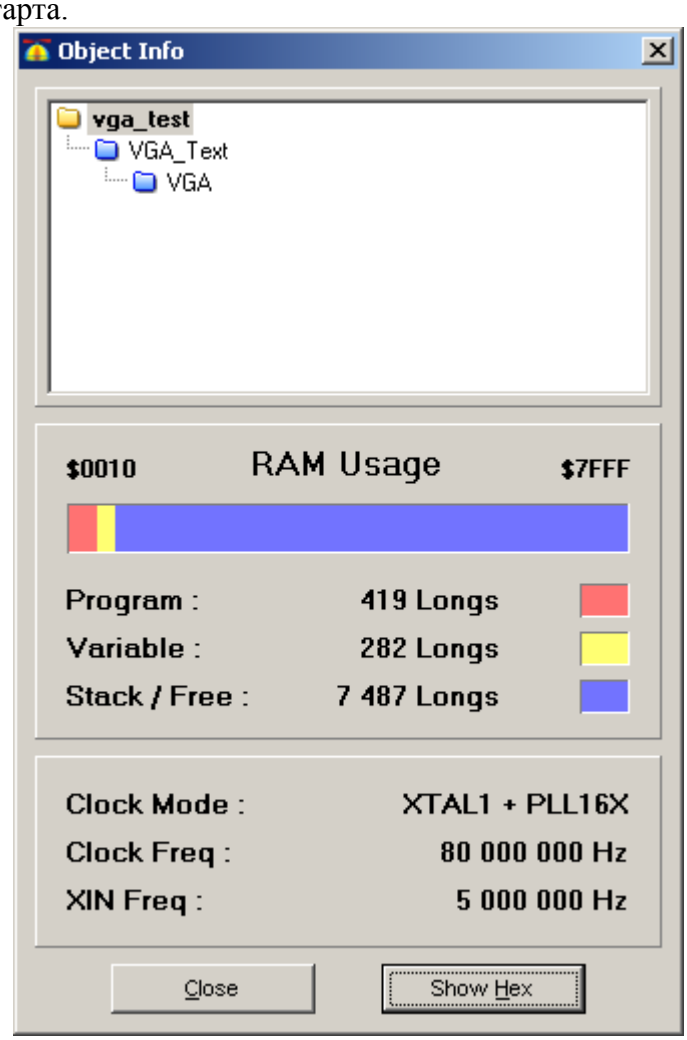

Рис. 6 – «Прожорливость» VGA

Теперь самое интересное: посмотрим сколько ресурсов потребляется. Результат компиляции представлен на рис. 6. Данное окно появляется после компиляции проекта (нажатие F8) и содержит основную информацию о микропрограмме (такую как размер частей, глобальную настройку тактирования и структуру модулей).

Как можно заметить, израсходовано (419+282)*4 = 2084 байт общей памяти. Довольно скромно для модуля, генерящего текстовый видеосигнал. Другое дело, размер текстовой области, всего 40*15 символов, так как используется встроенный качественный шрифт с размером символа 16*32. Если посмотреть исходники модуля VGA, то видно, так же задействовуется только одно дополнительное ядро (которое и занимается генерацией, основной же код продолжает работать на 1ом ядре).

При желании можно сгенерировать сигнал одновременно на целых 6-7 мониторов, главное чтобы хватило пинов ввода-вывода (а их хватит только на 3).

Раз уж заговорил о генерации сигнала на несколько мониторов, проведем эксперимент. Можно ли без особых плясок заставить стандартный модуль vga генерировать сигнал на несколько мониторов (хотя бы на 2)?

Пробуем такой код:

```
01| CON
02|   _clkmode = xtal1 + pll16x
03|   _xinfreq = 5_000_000
04|
05| OBJ
06|   text  : "vga_text"  ' monitor 1
07|   text2 : "vga_text" ' monitor 2
08|
09| PUB start | i
10|   text.start(16)
11|   text2.start(8)
12|
13|   i := 0
14|
15|   repeat
16|
17|     text.out($C)  ' set color
18|     text.out(i)
19|     text.str(string("Test 1. "))
20|
21|     text2.out($C) ' set color
22|     text2.out(32-i)
23|     text2.str(string("Test 2. "))
24|
25|     i++
26|     if i > 32
27|       i := 0
28|     waitcnt(CNT + 80_000_000 / 3)
```

Как можно заметить, модуль vga_text импортируется 2 раза под разными именами. Это позмоляет создать 2 его копии, имеющие разные экземпляры переменных и буферов. К сожалению, этот трюк дублирует и код самих функций, но, видимо, обойти такое никак нельзя (если только создавать копии на разных ядрах с отдельными стеками, что не особо приемлемо и требует существенного усложнения).

Асинхронный интерфейс (UART)

Теперь следующий модуль: асинхронный последовательный интерфейс. Рассмотрим модуль simple_serial.

```
01| CON
02|  _clkmode = xtal1 + pll16x
03|  _xinfreq = 5_000_000
04|
05| OBJ
06|   text    : "vga_text"
07|   serial  : "simple_serial"
08|
09| PUB start | ch
10|   serial.init(31, 30, 9600)
11|   text.start(16)
12|   repeat
13|     ch := serial.rx
14|     if ch == $0D
15|       text.out($0D)
16|     elseif (ch < 15) & (ch > 127)
17|       text.out("?")
18|     text.out(ch)
```

В данном коде происходит подключение модуля simple_serial, настройка его на пины ввода-вывода совпадающие с теми, что используются для программирования. Это позволяет не подпаивать новые соединения.

При надобности, по аналогии с двумя мониторами, можно реализовать до 7 программных модулей UART на любых пинах ввода-вывода.

Модуль simple_serial простой, задействует одно дополнительное ядро (для мониторинга приема по UART). Модуль целиком написан на Spin, поэтому довольно ограничен по скорости работы: максимальная допустимая скорость передачи данных 19200 бодд, что довольно скромно.

К сожалению, других модулей для реализации UART в Propeller Tool мною найдено не было. Если потребуется большая скорость работы, придется либо искать реализацию, либо писать самостоятельно с использованием ассемблера.

Ссылки

1. Актуальная версия документа:
   http://www.igorkov.org/pdf/propeller2.pdf
2. Русская докуменатция по чипу Propeller:
   www.parallax.com/dl/docs/prod/prop/PM-v1.0-RUS-v1.0.pdf
3. Простой пример генерации сигнала (меандр, 4МГц):
   http://www.igorkov.org/lst/propeller2_gen.spin
4. Пример (генерация VGA-сигнала):
   http://www.igorkov.org/lst/propeller2_vga.spin
5. Пример (генерация VGA-сигнала на 2 монитора):
   http://www.igorkov.org/lst/propeller2_vga2.spin
6. Пример (простая терминальная программа):
   http://www.igorkov.org/lst/propeller2_term.spin
7. Пример (терминальная программа в высоком разрешении):
   http://www.igorkov.org/lst/propeller2_term_hires.spin
8. Официальный форум Parallax по чипу Propeller:
   http://forums.parallax.com/forums/default.aspx?f=25