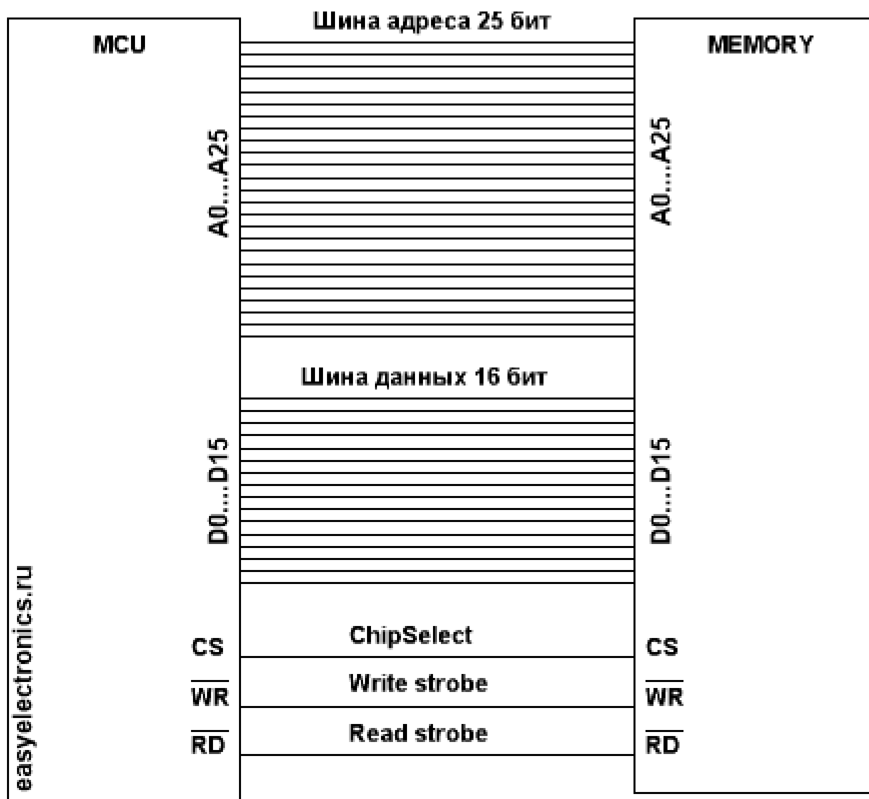
STM32. Контроллер внешней параллельной памяти FSMC

ARM. Учебный курс 20 сентября 2020 DI HALT 38 комментариев

У микроконтроллеров собственной памяти мало, даже если говорить о каком-нибудь жирном прежирном Cortex, все равно: как волка ни корми, а у медведя, т.е. полноценного компьютера, толще. Поэтому практически все микроконтроллеры, в своем жирном исполнении так или иначе позволяют подцеплять к себе внешнюю параллельную память. Даже древний, как говно мамонта, AT89C51 это умел. Что уж говорить про AVR и STM32.

Параллельная память



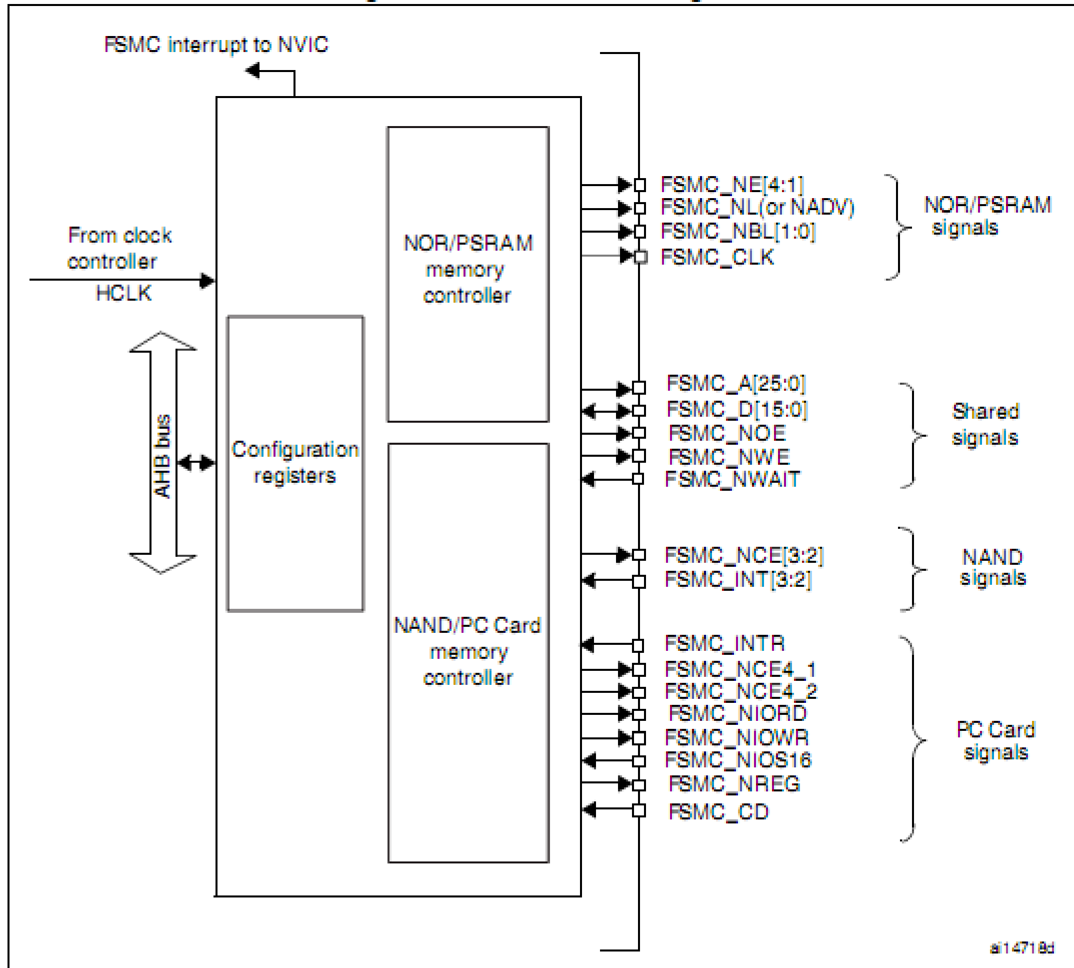
Для такой памяти характерны две черты: наличие двух шин: адреса и данных, и разных там стробов. На чтение, запись, тактовых и прочих. А вся суть работы с ними предельно простая. Чтение — мы выставляем на ногах адресной шины (коих обычно от 16 до 32) адрес нужной ячейки, отдельной ногой указываем, что у нас чтение, дергаем стробом и на шине данных (8 или 16 бит, обычно) появляется желанные байты. Запись похожим образом, только тут на шину данных мы выкладываем то, что хотим записать, на адресную шину кладем адрес куда надо записать, расставляем контрольную линию в режим записи и дергаем стробом. Опа, данные в памяти. Поскольку тут не требуется совершать сложных логических действий, то это все работает очень быстро, а реализовать можно даже на логике рассыпной. Естественно, что в разных МК такие интерфейсы были всегда.

FSMC

В STM32 за работу с внешней памятью отвечает такой блок как FSMC — Flexible Static Memory Controller. Который позволяет подключать к МК множество разных типов памяти (SRAM, NAND Flash, NOR Flash, PSRAM, PC Card и практически все что угодно, что имеет интерфейс схожий с i8080 или MC6800, например, LCD дисплей, с параллельной шиной. Причем автоматически будут переключаться банки памяти, если таковые нужны.

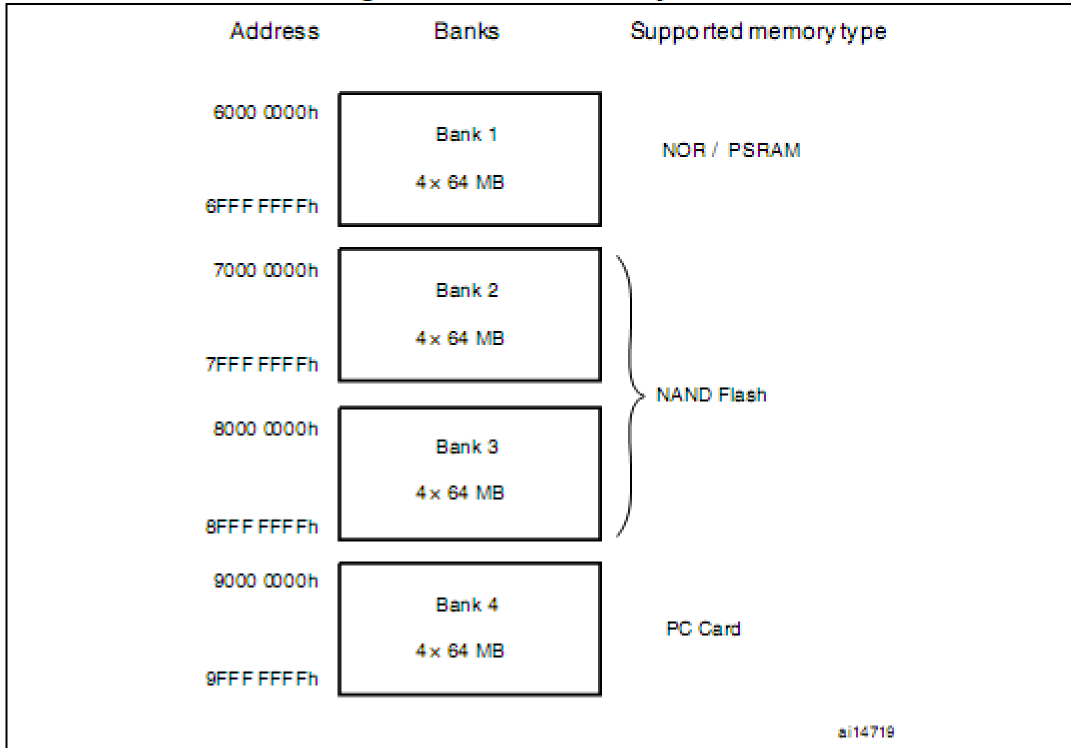
Со стороны софта же обращение к внешней памяти выглядит совершенно нативным образом, у STM32 огромное адресное пространство и неслабый кусок его выделен под работу с внешней памятью — адреса с 0x60000000 по 0x9FFFFFFF. Мы просто настраиваем контроллер и пишем в нужный сегмент этого адресного пространства. Как будто это наша собственная оперативная память. Банально, через указатели. А контроллер сам развернет это во внешний мир и запишет-считает из внешней микросхемы памяти.

Figure 185. FSMC block diagram

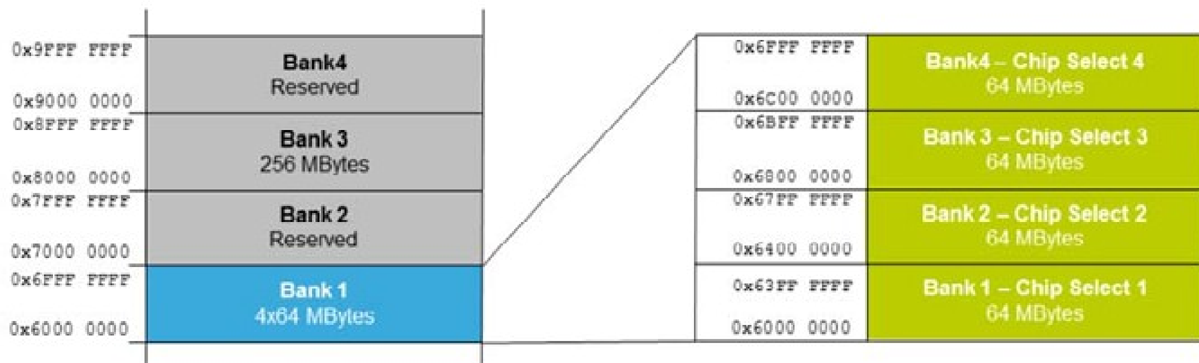


Фактически же ,контроллера там два. Один отвечает за работу с типом NOR/PSRAM, другой с NAND/PC Card у этих интерфейсов несколько различается логика работы, но они довольно похожи. При этом у них те сигналы, что совпадают для каждого типа тут общие, ну там шина данных, шина адреса, линии Write-Read, а какие то специфичные линии для каждого типа памяти раздельно свои. Что позволяет одновременно цепляться на одну шину разные девайсы и они могут работать вместе, разумеется попеременно.

Под каждый тип памяти свои адреса. При этом внутри они еще делятся на четыре банки, каждой из которой выделяется своя нога Chip Enable снаружи, а также свои настройки скорости и таймингов. Т.е. можно вообще сделать всратую солянку из разных микросхем, сконфигурировать и это будет работать.



Например, контроллер NOR/PSRAM может адресовать четыре микросхемы памяти по 64 мегабайта каждая. Они будут сидеть параллельно на шине данных и шине адреса, а также линиях строба чтения и записи. Но к каждой будет еще идти индивидуальная линия Chip Enable, от ног контроллера FMSC зовутся они NE1…NE4. (т.е. Not Enable1.. Жутко меня вымораживает эта нотация, писать инверсное состояния бита буквой N перед именем. Ну хоть не заглавной бы писали, как это микрочип делает, nE1 всяко читабельней чем NE1. А лучше надчеркиванием, как это делал всегда ATMEL, бесит!)



При этом уровень на ногах NE1…NE4 не нужно ставить вручную, он зависит от адреса банки в которую мы пишем, и ставится автоматически. А конкретней, зависит от битов адреса [27:26]. Т.е. изнутри нам эти четыре микросхемы будут выглядеть как единое, плоское адресное пространство, мы в него пишем/читаем свободно, а контроллер сам подключает нужную микросхему. При этом тайминги настраиваются для каждой микросхемы отдельно! То есть они могут быть разными. Красота же!

Выводы той части контроллера, что NOR/PSRAM там в даташите расписаны прям отдельно.

Table 105. Nonmultiplexed I/O NOR Flash

FSMC signal name	I/O	Function
CLK	O	Clock (for synchronous access)
A[25:0]	O	Address bus Шина адреса
D[15:0]	I/O	Bidirectional data bus Шина данных
NE[x]	O	Chip select, x = 1..4 Выходы с банок
NOE	O	Output enable Линия RD
NWE	O	Write enable Линия WR
NL(=NADV)	O	Latch enable (this signal is called address valid, NADV, by some NOR Flash devices)
NWAIT	I	NOR Flash wait input signal to the FSMC

▌Ошибки

Также этот контроллер может генерировать ошибки, которые приводят к исключению Hard Fault.

Ошибки возникают в следующих случаях:

- Когда мы пишем или читаем из адресного пространства FSMC, но оно не включено. Т.е. мы пытаемся обращаться в пустоту. Это, кстати, характерно не только для FSMC, но и для многих других блоков.
- Когда мы обращаемся к NOR Flash памяти, но бит FACCEN регистра FSMC_BCRx для конкретного банка сброшен. Напомню, что банок там четыре.
- Когда мы обращаемся к PC Card, а пин ее физического наличия не выставлен.

И на этот раздел прошу обратить особое внимание, т.к. можно просто забыть где-то поставить битик, или подтянуть ножку, а потом долго пытаться понять какого хрена у нас все крашится. Если же стучаться туда пытается DMA, то это приводит к тому, что DMA канал автоматически отключается. Так что если у вас вдруг стал не с того ни с сего выключаться DMA проверьте адреса и наличие получателя на том конце.

▌Пример

Вообще работать с этим контроллером довольно просто. Главное не запутаться в множестве ног. Подключать будем дисплей на контроллере **RA8875L3N**. Стандартная довольно шняга, дисплей 800×480, 7 дюймов диагональ. С тачскрином. Умеет и режим шина типа i8080 и M68, а еще spi и i2c. В общем, навороченная зверюшка. Нам же нужен i8080. По такому же протоколу работает и дисплей на **SSD1963**, разница там может быть разве что в командах инициализации.

В интернете полно примеров подключения этой шняги, но везде все сводится к тому, что а давайте мы запустим куб, натыкаем галочек, впишем сюда вот эти магические цифры и у нас будет магия. Порочная практика, из-за нее хер найдешь теперь нормальный пример с подробностями, все засрано этим кубом. Который еще и генерит лютую кашу в виде проекта, нет бы давал снипеты для копипаста ,то было бы куда ни шло. А так… БУЭЭЭЭЭ…

Нахер магию, нахер куб, нахер HAL и даже SPL, подключать будем по даташиту с подробным описанием что, откуда, ручками, на регистрах. Тем более подключать то там нечего.

Выводы? С выводами все довольно просто, ищем спецификацию интерфейса и узнаем, что у нас тут есть:

- **D0…D15** — шина данных.
- **CS** — Активация чипа ака Chip Select
- **RS** — Данные/команда. Определяет что будет на линии данных, команда или данные изображения
- **RST** — Сброс контроллера дисплея
- **WR** — Строб записи. Выставляем на линию данных байт и дергаем этой ногой для записи в дисплей
- **RD** — Строб чтения. Дергаем этой ногой и дисплей выдает байт на шину данных, который мы потом читаем
- **INT, TINT** — выходы прерывания. Через него дисплей может пнуть наш контроллер на предмет внешних событий, например нажатие на тачскрин. В данном случае не нужен.

Так как у нас интерфейс i8080, то подключаться мы будем как NOR/PSRAM устройство к соответствующему контроллеру.

Подключается это все следующим образом.

- **D0… D15 — FMSC_D0…D15**. Шина данных, тут все просто, линия к линии. Можно и по 8 битному формату подключить. дисплей такое вроде поддерживает, но обновляться картинка будет ну очень уж медленно. Считай три цвета надо будет протащить за три операции записи. А тут все в одной пролезет.
- **RS — FMSC_Ax** Адресной шины у нас нет, дисплей подключается не как микросхема памяти (а жаль, было бы круто). Зато надо переключать режим данные/команда. Лучше всего для этого применить один из адресных выводов шины. И

тогда получится, что для записи команды нам надо будет закинуть их по адресу X, а для записи данных по адресу X+1 и все.

Но тут есть одно очень и очень неоднозначное западло.

Дело в том, что внешняя адресация FMSC зависит от разрядности шины данных (RM00088 стр. 511 табл. 101). Если у нас разрядность шины данных составляет 8 бит, то все нормально, адрес в адресном пространстве равен адресу на линиях данных. Т.е. если банк начинается с адреса 0x60000000, то при записи в 0x60000000 A0=0, а запись в 0x60000001 сделает линию A0 = 1. Т.е. младший бит адресной шины соответствует младшему биту адреса.

Но если мы переводим шину в 16 разрядный режим, то у нас включается внешняя адресация в словах.

Выглядит это как сдвиг адреса вправо перед тем как он попадет на шину адреса Ax. Т.е. адрес делится на два, оно и логично, данные то у нас уже словами пошли. И тогда для того, чтобы поднять A0 надо записать не по адресу 0x60000001 (это будет всего лишь адрес второго байта первого слова, что был на шине данных при адресе 0x60000000), а по адресу следующего слова, т.е. 0x60000002 и тогда у нас выйдет бит на A0. То же касается, естественно, и других линий A0 в 16 разрядном режиме.

Запомните этот момент. Т.к. однажды, решив подключить дисплей по 8 битной шине, ну чтобы ноги сэкономить, вы лихо сломете мозги себе, пытаясь понять почему работающий код перестал работать.

- **CS — FMSC_NEx** Выбор чипа, ака Chip Select берется от той банки с которой удобней работать или разводить плату. Разница лишь в адресе куда мы будем писать/читать данные и по которому мы будем дергать линию Ax. Т.е. если у нас для дрыга RS взята A0 и надо вычислить адрес куда же нам валить данные, то пляшем от ноги NE. За NE1 отвечает банк 1, с началом в 0x60000000, а за ногу NE2 уже банк 2, с началом в 0x64000000 и соответственно у вас будет адрес для данных 0x64000002. Аналогично для банка 3 и 4. Можно выбрать любой, единственное, что это не всегда возможно, но об этом будет ниже.
- **RD — FMSC_NOE** Линия строба чтения идет на Output Enable (почему его переименовали с RE или RD вдруг я хз, неужто патентные войны, как было с i2c когда то?)
- **WR — FMSC NWE** Линия строба записи. Идет на Write Enable
- **INT — GPIO** Линии сброса дисплея, а также линии прерываний от дисплея и тача идут на любые удобные порты GPIO тут вообще без разницы.
- **TINT — GPIO**
- **RST — GPIO**

Что ж, попробуем все это привести ближе к теме. Контроллером будет STM32F103VGT6, у меня он просто есть уже разведенный, на готовой плате, взял его с платы одного своего текущего проекта. У него 100-ногий корпус, а в этом случае у нас FMSC сильно кастрированный. У него нет полноценной адресной шины, от линий адреса A0…A25 остались только жалкие огрызки — адресные линии A16…A23. Ну и, соответственно, линия NE выведена только у одного банка, у первого. Okay :(

Настроим GPIO, у меня используется самописная настройка портов в виде матрицы. Про нее я уже писал как то раз. Так что вся настройка портов в одном месте и выглядит так:

```
1   const tGPIO_Line IOs[] =      {
2                                 { GPIOB, 0,  OUT_10MHz + OUT_PP, LOW},          // Reset
3                                 { GPIOA, 7,  IN + IN_PULL, HIGH},               // Int Line
4                                 { GPIOD, 0, OUT_50MHz + OUT_APP, LOW},          //
5                                 { GPIOD, 1, OUT_50MHz + OUT_APP, LOW},          //
6                                 { GPIOD, 4, OUT_50MHz + OUT_APP, LOW},          // RD - NOE
7                                 { GPIOD, 5, OUT_50MHz + OUT_APP, LOW},          // WR - NWE
8                                 { GPIOD, 7, OUT_50MHz + OUT_APP, LOW},          // CS -NE1
9                                 { GPIOD, 8, OUT_50MHz + OUT_APP, LOW},          //
10                                { GPIOD, 9, OUT_50MHz + OUT_APP, LOW},          //
11                                { GPIOD, 10, OUT_50MHz + OUT_APP, LOW},         //
12                                { GPIOD, 11, OUT_50MHz + OUT_APP, LOW},         // RS - A16
13                                { GPIOD, 14, OUT_50MHz + OUT_APP, LOW},         //
14                                { GPIOD, 15, OUT_50MHz + OUT_APP, LOW},         //
15                                { GPIOE, 7, OUT_50MHz + OUT_APP, LOW},          //
16                                { GPIOE, 8, OUT_50MHz + OUT_APP, LOW},          //
17                                { GPIOE, 9, OUT_50MHz + OUT_APP, LOW},          //
18                                { GPIOE, 10, OUT_50MHz + OUT_APP, LOW},         //
19                                { GPIOE, 11, OUT_50MHz + OUT_APP, LOW},         //
20                                { GPIOE, 12, OUT_50MHz + OUT_APP, LOW},         //
21                                { GPIOE, 13, OUT_50MHz + OUT_APP, LOW},         //
22                                { GPIOE, 14, OUT_50MHz + OUT_APP, LOW},         //
23                                { GPIOE, 15, OUT_50MHz + OUT_APP, LOW}          //
24                                                              };
```

Тут все довльно просто, сразу указано как настроена нога, поясню только что что OUT_APP, означает режим альтернативной функции (**A**FIO **P**ush**P**ull), с подтяжкой в LOW. Настраивайте это как хотите у себя, хоть через HAL, хоть через spl, хоть LL.

Также, надо настроить тактирование периферии:

```
1   RCC->APB2ENR    |= RCC_APB2ENR_IOPAEN;  // Тактирование портов
2   RCC->APB2ENR    |= RCC_APB2ENR_IOPBEN;
3   //RCC->APB2ENR  |= RCC_APB2ENR_IOPCEN;
4   RCC->APB2ENR    |= RCC_APB2ENR_IOPDEN;
5   RCC->APB2ENR    |= RCC_APB2ENR_IOPEEN;
6   //RCC->APB2ENR  |= RCC_APB2ENR_IOPFEN;
7   //RCC->APB2ENR  |= RCC_APB2ENR_IOPGEN;
8   RCC->APB2ENR    |= RCC_APB2ENR_AFIOEN;  // Включаем AFIO
9   RCC->AHBENR     |= RCC_AHBENR_FSMCEN;   // Включаем FSMC
10
11
```

```
12   AFIO->MAPR &=~AFIO_MAPR_SWJ_CFG_Msk;            // Делаем ремап линий JTAG, точнее включаем JTAG, оставляем только SWD
13   AFIO->MAPR |= AFIO_MAPR_SWJ_CFG_JTAGDISABLE;    // Иначе он попадает на ноги.
```

Теперь определяем наш базовый адрес. Для удобства сведено все в структуру. Сидеть все будет так, CS на NE1, нога RS на A16

```
1  typedef struct
2  {                                   // Структура адресуется по порядку. Базовый адрес в начале.
3          volatile uint16_t LCD_RAM;      // 0x6001FFFE  >> 1 = 0x0000FFFF bit 16 = 0 т.е. DATA
4          volatile uint16_t LCD_REG;      // 0x6001FFFE + 2 (т.к. uint16) = 0x60020000 >> 1 = 0x00010000 bit 16 = 1 т.е. CMD
5  } LCD_TypeDef;
6
7  /* LCD is connected to the FSMC_Bank1 and NE1 is used as ship select signal, A16 use as RS */
8  #define LCD_BASE    ((uint32_t)(0x60000000 | 0x0001fffE))
9  #define LCD         ((LCD_TypeDef *) LCD_BASE)
```

Я себе вот такой вот кусочек заготовил и если надо копирую его себе и правлю под нужную задачу. Никаких структур, никаких левых функций. Все вот, на виду. Заодно добавил в комменты ряд выдержек из даташита.

```
1   #define FSMC_Bank1_NORSRAM1     ((uint32_t)0x00000000)
2   #define FSMC_Bank1_NORSRAM2     ((uint32_t)0x00000002)
3   #define FSMC_Bank1_NORSRAM3     ((uint32_t)0x00000004)
4   #define FSMC_Bank1_NORSRAM4     ((uint32_t)0x00000006)
5   #define FSMC_Bank2_NAND         ((uint32_t)0x00000010)
6   #define FSMC_Bank3_NAND         ((uint32_t)0x00000100)
7   #define FSMC_Bank4_PCCARD       ((uint32_t)0x00001000)
8
9       FSMC_Bank1E->BWTR[FSMC_Bank1_NORSRAM1] = 0x0FFFFFFF;
10
11                                                                     // BCR&BTR Register see RM000008 p. 541
12      FSMC_Bank1->BTCR[FSMC_Bank1_NORSRAM1] =
13                                  0 << FSMC_BCRx_CBURSTRW_Pos   |       // write 0 - async 1 - sycnc
14                                  0 << FSMC_BCRx_ASYNCWAIT_Pos |       // Wait signal during asynchronous transfers
15                                  0 << FSMC_BCRx_EXTMOD_Pos     |       // Extended mode enable. Use BWTR register or no
16                                  0 << FSMC_BCRx_WAITEN_Pos     |       // Wait enable bit.
17                                  1 << FSMC_BCRx_WREN_Pos       |       // Write enable bit
18                                  0 << FSMC_BCRx_WAITCFG_Pos    |       // Wait timing configuration. 0: NWAIT signal is active one data cycle
19                                                                        // before wait state 1: NWAIT signal is active during wait state
20                                  0 << FSMC_BCRx_WRAPMOD_Pos    |       // Wrapped burst mode support
21                                  0 << FSMC_BCRx_WAITPOL_Pos    |       // Wait signal polarity bit. 0: NWAIT active low. 1: NWAIT active high
22                                  0 << FSMC_BCRx_BURSTEN_Pos    |       // Burst enable bit
23                                  1 << FSMC_BCRx_FACCEN_Pos     |       // Flash access enable
24                                  1 << FSMC_BCRx_MWID_Pos       |       // 0 = 8b 1 = 16b
25                                  2 << FSMC_BCRx_MTYP_Pos       |       // 0 = SRAM 1 = CRAM 2 = NOR
26                                  0 << FSMC_BCRx_MUXEN_Pos      |       // Multiplexing Address/Data
27                                  1 << FSMC_BCRx_MBKEN_Pos;             // Memory bank enable bit
28
29
30
31      FSMC_Bank1->BTCR[FSMC_Bank1_NORSRAM1 + 1] =
32                                  0 << FSMC_BTRx_ADDSET_Pos   |    // Address setup phase duration 0..F * HCLK
33                                  0 << FSMC_BTRx_ADDHLD_Pos   |    // Address-hold phase duration 1..F * 2 * HCLK
34                                  1 << FSMC_BTRx_DATAST_Pos   |    // Data-phase duration 1..FF * 2 * HCLK
35                                  0 << FSMC_BTRx_BUSTURN_Pos  |    // Bus turnaround phase duration 0...F
36                                  1 << FSMC_BTRx_CLKDIV_Pos   |    // for FSMC_CLK signal 1 = HCLK/2, 2 = HCLK/3 ...   F= HCLK/16
37                                  0 << FSMC_BTRx_DATLAT_Pos   |    // Data latency for synchronous NOR Flash memory 0(2CK)...F(17CK)
                                    0 << FSMC_BTRx_ACCMOD_Pos;       // Access mode 0 = A, 1 = B, 2 = C, 3 = D Use w/EXTMOD bit
```

Теперь немного поясню. Т.к. контроллер у нас NORSRAM используется, то настраивать и включать будем его. Банк у нас 1, без вариантов. В CMSIS все это определено как то коряво, как массив, где регистры FSMC_BTR и FSMC_ BCR никак не разделены, а слиты в кучу, словно они поленились. Приходится их индексным методом доставать, благо они парами идут в памяти, друг за другом. поэтому для BCR1 адрес от базового FSMC_Bank1_NORSRAM1, а для BTR адрес FSMC_Bank1_NORSRAM1+1.

Дальше там биты, сделал как в старом добром AVR, я так люблю :))))

Начинаем настраивать. Сначала определимся в каком режиме все должно работать. Смотрим времянки на интерфейс и на то, что контроллер нам предлагает. В нашем случае это NOR Flash или Mode 2.

Смотрим табличку 114 в RM00008

Table 114. FSMC_BCRx bit fields

Bit number	Bit name	Value to set
31-20	Reserved	0x000
19	CBURSTRW	0x0 (no effect on asynchronous mode)
18:16	Reserved	0x0 (no effect on asynchronous mode)
15	ASYNCWAIT	Set to 1 if the memory supports this feature. Otherwise keep at 0.
14	EXTMOD	0x1 for mode B, 0x0 for mode 2
13	WAITEN	0x0 (no effect on asynchronous mode)
12	WREN	As needed
11	WAITCFG	Don't care
10	WRAPMOD	0x0
9	WAITPOL	Meaningful only if bit 15 is 1
8	BURSTEN	0x0
7	Reserved	0x1
6	FACCEN	0x1
5-4	MWID	As needed
3-2	MTYP[0:1]	0x2 (NOR Flash memory)
1	MUXEN	0x0
0	MBKEN	0x1

Определиться надо всего с парой бит. Все остальное или предрешено или не имеет значения :)

- **ASYNCWAIT** будет ли использоваться вход для сигнала занятности памяти. В нашем случае нет, у нас на дисплее есть нога WAIT, но она более выскоуровневая. Так что там 0.
- **EXTMOD** — расширенный режим. Если включить его, то мы можем задавать раздельные тайминги как для чтения, так и для записи. В этом случае регистр BWTR[FSMC_Bank1_NORSRAM1] будет содержать тайминги на запись, а обычный FSMC_BTR тайминги на чтение. Если же бит расширенного режима выключен, то FSMC_BTR содержит тайминги для всех вариантов данного банка.
- **WREN** — мы будем писать в дисплей, так что конечно этот бит нужен. Он включает ногу строба NWE.
- **MWID** — определяет ширину шины данных. Я решил применить 16 бит, так быстрей. Но можно сделать и 8 бит — сэкономите ноги, но помните про сдвиг адресации!!!

Теперь надо определиться с таймингами записи и чтения. У нас за них отвечает регистр FSMC_BTR1. И нам в Mode 2 доступны только две настройки:

Mode 2/B - NOR Flash

Figure 191. Mode2 and mode B read accesses

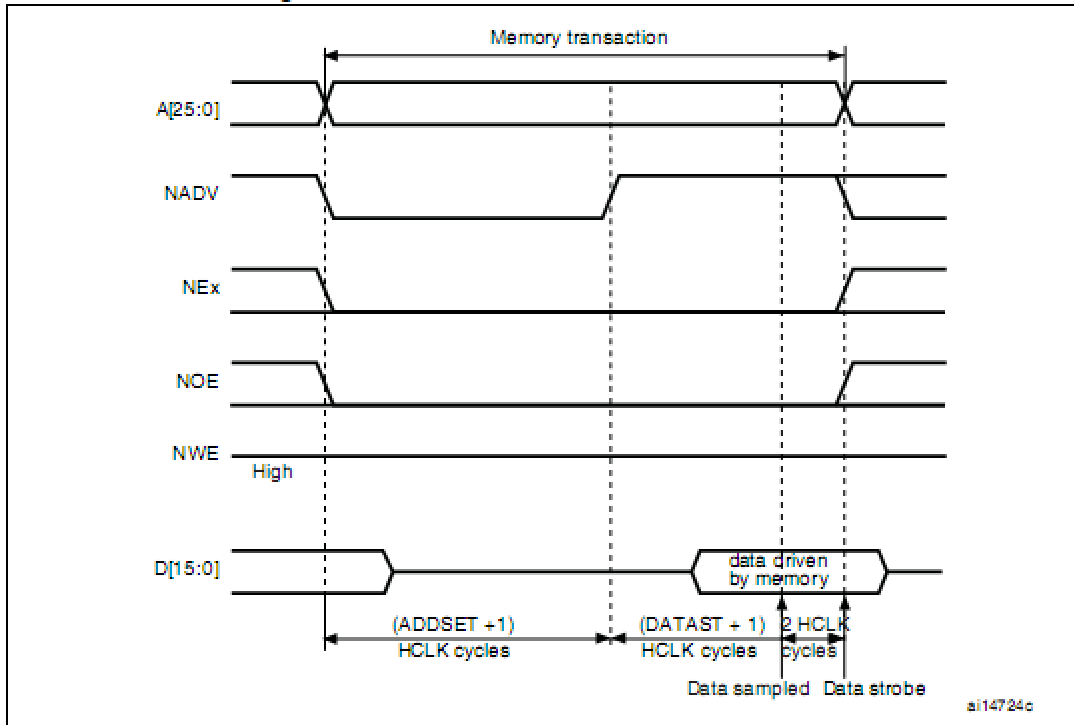
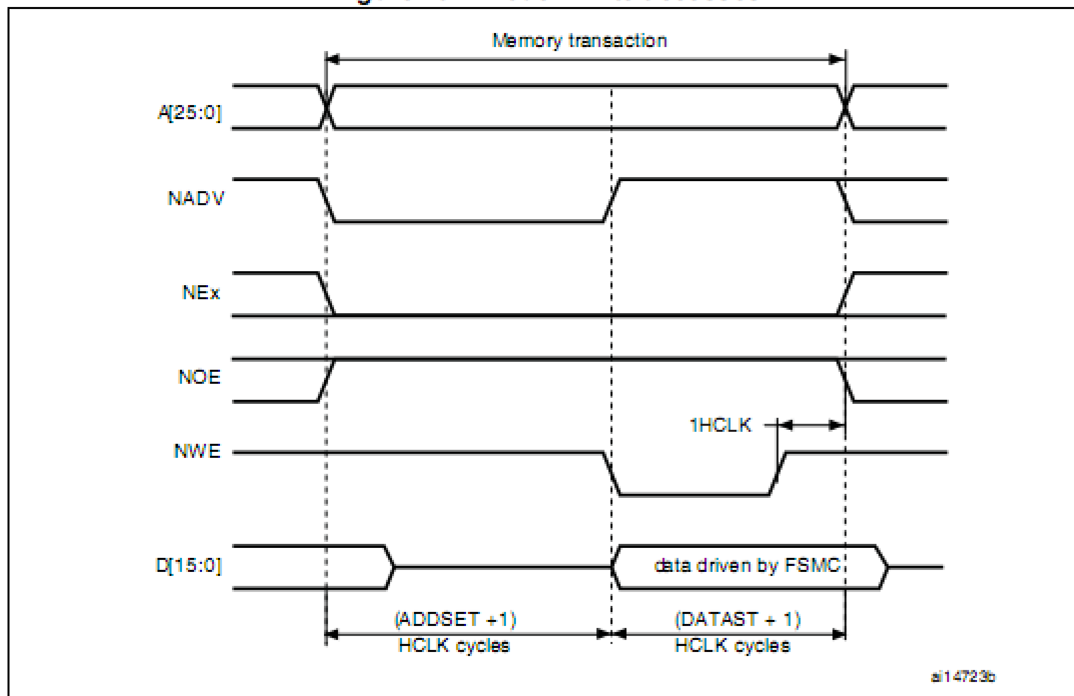


Figure 192. Mode2 write accesses



Смотрим табличку 115 в RM00008

Table 115. FSMC_BTRx bit fields

Bit number	Bit name	Value to set
31:30	Reserved	0x0
29-28	ACCMOD	0x1
27-24	DATLAT	Don't care
23-20	CLKDIV	Don't care
19-16	BUSTURN	Time between NEx high to NEx low (BUSTURN HCLK)
15-8	DATAST	Duration of the second access phase (DATAST+3 HCLK cycles) for read accesses. This value cannot be 0 (minimum is 1).
7-4	ADDHLD	Don't care
3-0	ADDSET[3:0]	Duration of the first access phase (ADDSET+1 HCLK cycles) for read accesses.

- **BUSTURN** нам не нужен, это для пакетной записи в PSRAM.
- **DATAST** время за которое данные устаканятся на шине данных и дисплей готов будет их считать.
- **ADDSET** время за которое установится и будет воспринят адрес на адресной шине. Т.к. у нас адресная шина рулит линией RS, то надо смотреть время от RS до готовности считать данные.

Смотрим даташит на дисплей:

8080 – 8/16-bit Interface

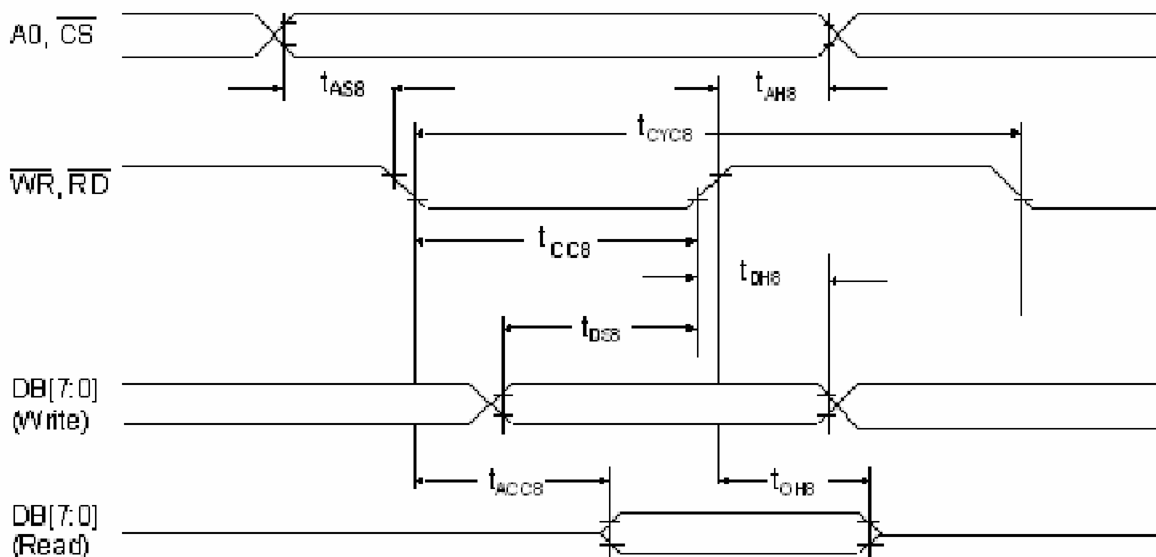


Table 6-2 : 8080 MCU I/F Timing

Symbol	Parameter	Rating		Unit	Symbol
		Min.	Max.		
t_{CYC8}	Cycle time	50	--	ns	tc is one system clock period: tc = 1/SYS_CLK
t_{CC8}	Strobe Pulse width	20	--	ns	
t_{AS8}	Address setup time	0	--	ns	
t_{AH8}	Address hold time	10	--	ns	
t_{DS8}	Data setup time	20	--	ns	
t_{DH8}	Data hold time	10	--	ns	
t_{ACC8}	Data output access time	0	20	ns	
t_{OH8}	Data output hold time	0	20	ns	

Картинку конечно рисовал наркоман. Все в кашу. У них там чтоль вообще нормоконтроля нет? Наша Геннадьевна бы из за такой чертеж выдрала без вазелина шваброй.

Время от активации чипа через CS до готовности ловить строб совпадает со временем установки RS, его тут назвали A0, видимо подразумевая, что эта линия всегда будет на адресной линии A0. Говорю же, наркоманы. Это время tAS8 и табличка гласит, что оно может быть от 0. Т.е. на запись ADDSET можно смело ставить ноль.

А вот время от начала движухи на линии DB0..7 и до готовности, на перегибе строба WR обозначено как tDS8 И оно у нас тут 10ns для записи.

Для чтения же время от того как у нас упадет линия строба RD, и тем самым даст понять дисплею, что с него хотят читать равно tACC8 и составляет от 0 до 20ns. Берем по максимуму 20ns.

Чтобы не заморачиваться с настройкой разных таймингов для чтения и записи через EXTMOD примем их по максимальному значению 20ns. А теперь давайте думать сколько записать в DATAST. Из графика 191 узнаем, что длина этих циклов это DATAST+1 тик HCLK

FSMC тактуется от шины HCLK, которая сидит после предделителя AHB. У меня частота настраивается вручную и я точно знаю, что у меня там 72МГц. Вам же этот вопрос надо будет выяснить. А раз так, то один тик HCLK = 1/72000000 = 1.4E-8 т.е. около 14ns т.е. уже больше тайминга. А нам надо 20ns. Т.е. в DATAST запишем 1, будет даже немного с запасом.

Теперь код инициализации FSMC будет предельно понятен.

Осталось написать функции записи команды и данных. МЫ просто берем ранее сделанную структуру с адресами и просто пишем туда данные:

```
1    void LCD_CmdWrite(const uint16_t Command)
2    {
3            LCD->LCD_REG = Command;
4    }
5
6    void LCD_DataWrite(const uint16_t Data)
7    {
8            LCD->LCD_RAM = Data;
9    }
10
11   void LCD_WriteReg(const uint8_t LCD_Reg, const uint16_t LCD_RegValue)
12   {
13           LCD->LCD_REG = LCD_Reg;
14           LCD->LCD_RAM = LCD_RegValue;
15   }
```

Все, готово. Можно делать контроллеру дисплея сброс и начинать кормить дисплей командами инициализации:

```
1            IO_SetLine(io_TFT_Reset, HIGH);
2            delay_ms(5);
3
4            LCD_WriteReg(0x01, 0x01); //PWRR: LCD on, sleep off, reset ON
5
6            delay_ms(1);
7
8            LCD_WriteReg(0x01, 0x00); //PWRR: LCD on, sleep off, reset OFF
9
10           delay_ms(1);
11
12           LCD_WriteReg(0x88, 0x0B); //PLL Control Register 1
13
14           delay_ms(1);
15
16           LCD_WriteReg(0x89, 0x02); //PLL Cntrol Register 2
17
```

Инициализация самого дисплея и работа с ним требует отдельной статьи. Там целый мир. Адреса, встроенные функции рисования графики, разные режимы цветов… ой дофига всего. Я ее тоже планирую написать, но по нему примеров масса, даташит подробный, бери да гони адреса в память, а FSMC вам в помощь.

Собранный проект на FreeRTOS который ничего не делает, только инициализирует дисплей и заливает его туда сюда разными цветами прилагаю. Там все проще некуда.

- Проект для EmBitz но использует GCC toolchain и никаких сторонних библиотек, кроме CMSIS. Так что легко соберется в любой среде.