**Lesson 62**

**Part 1**

**FMC SDRAM**

Today we will try to work with the SDRAM memory chip.

SDRAM is a dynamic random access memory, the active element of each cell of which is a capacitor. Also, access to memory cells is not in a flat space, but through a matrix consisting of rows and columns. In each cell of this memory, 1 bit of info is stored. Compared with a static memory option, this memory has a very high speed, and very little power consumption.

The representative of such memory we will choose the microcircuit  **MT48LC4M32B2** from the company **Micron** .

This memory is installed on the **STM32F746G-DISCO** board , which we are very familiar with.

This memory has a capacity of 128 megabits, but since 32 of the data exchange contacts are uninstalled in order to save only 16, in this version there will be 64, which is also quite a lot. Physically, you get 8 megabytes of fast RAM.

For the convenience of accessing memory, since the algorithm for controlling the legs of such chips is not simple, **ST Microelectronics has** connected this memory using **FMC** technology , which allows you to access this memory using a flat address space, without worrying about the levels of multiple control and address legs.

Nevertheless, we will still get a little acquainted with the organization of work with this memory.

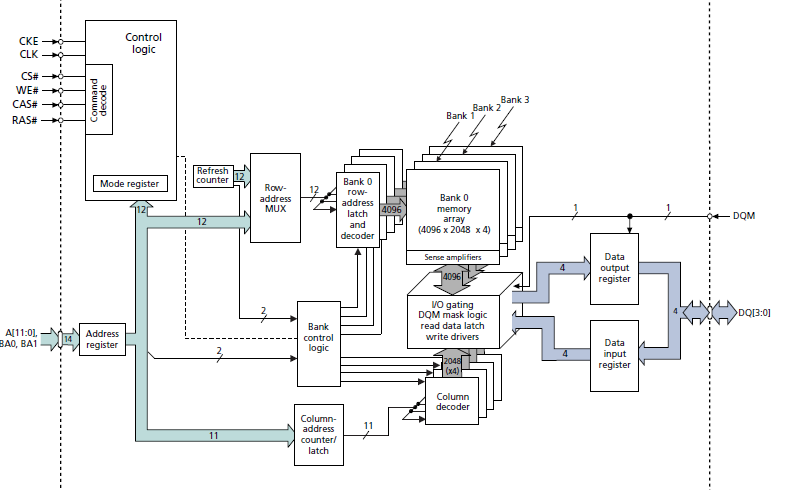
To do this, we will open the technical documentation for this chip and, firstly, see the pinout and assignment of the individual contacts of the chip. Only this later. First, some technical characteristics, most, so to speak, basic.

The memory configuration has a breakdown into 4 banks: 1 megadat by 32 bits (1 bit per data bus contact) and 4 banks, so we get 128 megabits.

1 megadres we get from the matrix calculation of 4096 rows of 256 columns.

Here in general, briefly on the organization of memory.

Let's see the flowchart of our microscheme



Here we see the organization of memory more clearly. Also we see here all the buses, except for the data bus. Rather, it is, but somehow strangely written.

The data bus consists of 32 legs - these are the legs of DQ0-DQ31. On these legs, exposing them at a certain time in a certain state, we write the information into memory, and also read it from them also at a certain time, removing their states.

The **DQM** feet **[3: 0]** are responsible for certain access to the data. We do not consider in detail.

The address bus consists of 12 legs A0-A11. We address it to specific addresses to certain memory cells, passing the address of the column and the address of the line.

Also to the address register are also the legs of the selection of one of the four memory banks BA0-BA1.

Also from the top we still see a number of legs related to the command logic.

**CLK** is a leg that provides synchronization, in fact, it is also called for many devices that require synchronization.

**CKE** is the leg that allows synchronization.

**CS** - the selection foot or Chip Select, necessary for selecting a certain chip, when there are several.

**WE** - Write Enable, responsible for the state of the record or read.

**CAS** - when this **pin** is activated, the status of the address bus is evaluated by the microchip as the address of the column.

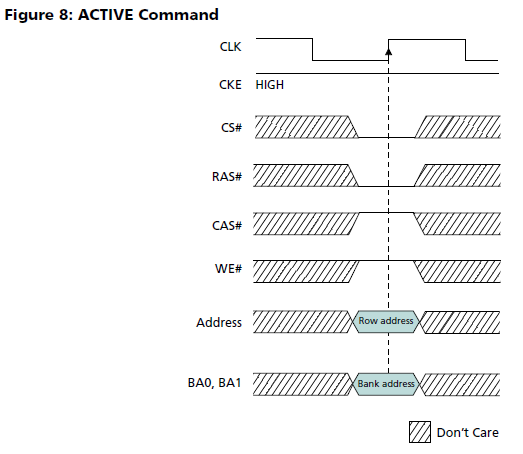
**RAS** - when this **pin** is activated, the address bus status is regarded as the address of the line.

There are also a number of instructions for configuring the chip, which we may get acquainted with in the programming process, but perhaps we will not need this, since all low-level functions will be **taken** over by **FMC** .

You can also get a little acquainted with how to read and write data at all. I think many will be interested. Although of course you can find all this by reading the technical documentation.

First, let's look at some commands.

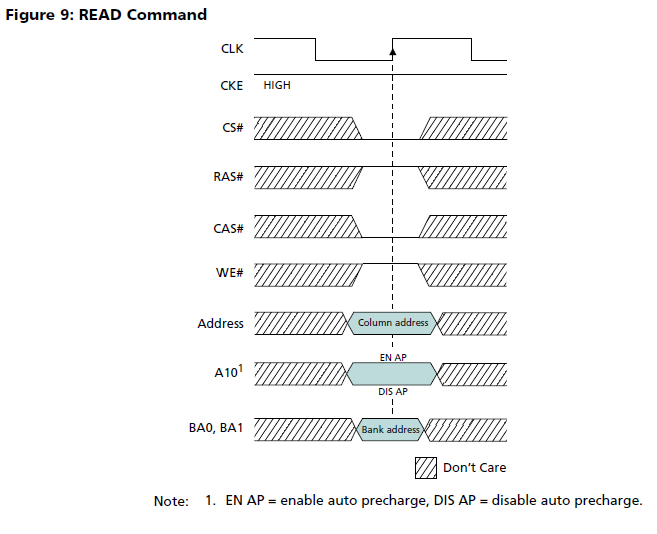
Activation command



This command is initialized to select a specific bank and the address of the string.

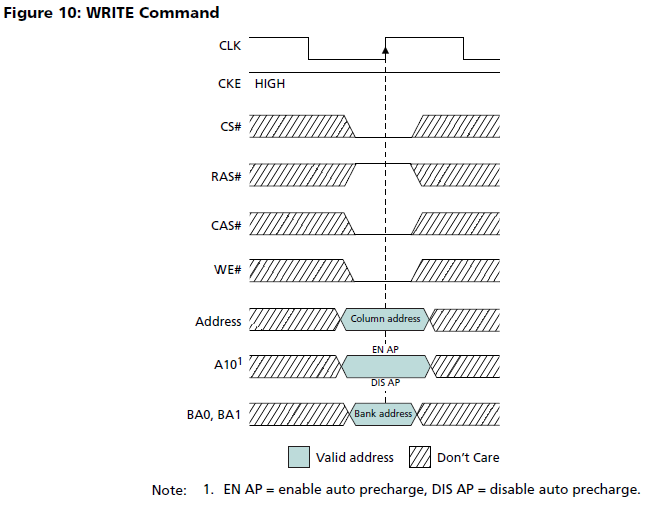
The next team after it will already tell the memory of whether we read it or write to it, and will also pass the address of the column.

Reading command



Here we see the necessary state of the legs.

Write command



Here we also see about the same picture, the only difference is the WE foot in the low state. That's the state of the chip and will understand what we write in it. In the last two operations, the address of the column is already transferred.

But this is of course only preliminary preparation for recording or reading. Further on, the data bus and the mass of variants of various types of access already enter into operation - in a row or random and several other interesting options, which are also described in the technical documentation that will be attached to the lesson at the end of the page.

From time to time we will still look at both the microcircuit and the debug board, and of course we will also refer to the debugging circuit diagram.

In the [**next part of the**](http://narodstream.ru/stm-urok-62-fmc-sdram-chast-2/) lesson, we are already working hard to create a project to solve our task - working with SDRAM memory using an FMC memory controller.

**Lesson 62**

**Part 2**

**FMC SDRAM**

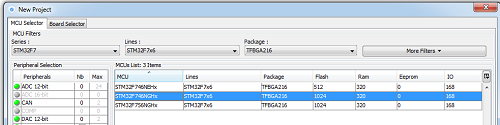
In the last part of our session, we met with a memory chip SDRAM  **MT48LC4M32B2** from the company **Micron's** .

We also got acquainted with the organization of this memory, how to address it, so that it began recording and reading.

Today we will continue the business started.

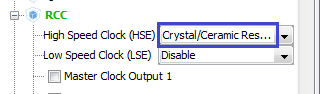
We will begin this part with the creation of the project.

We will create a new project in the Cube MX, select the controller (click on the image to increase the size)

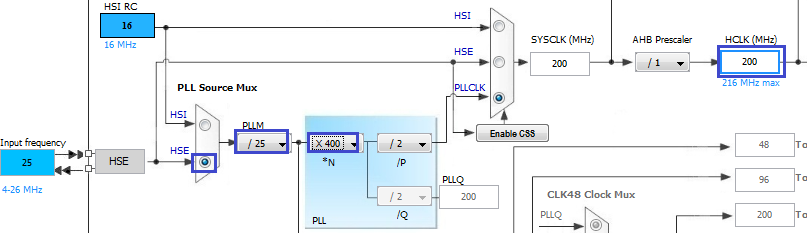
[](http://narodstream.ru/wp-content/uploads/2016/12/Image04-5.png)

Let's start configuring the project.

First of all, we turn on the quartz resonator



Then configure the Clock Configuration at 200 MHz as follows (click on the image to enlarge the image)

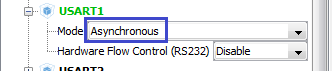
[](http://narodstream.ru/wp-content/uploads/2016/12/Image07-5.png)

We turn on the PI1 leg, which controls the green LED, to the output.

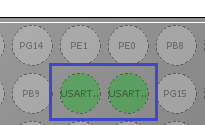
We will write a program that will first write certain data. starting with a specific address in the SDRAM, and then reading them from there to the same array. And we will look through the USART through the read data.

I think many people know, but not all. In the F746-DISCO, the USART interface can be used directly via the ST-Link programming port without connecting any additional adapters. The most important thing is to correctly configure it in Cube MX.

Include there USART1

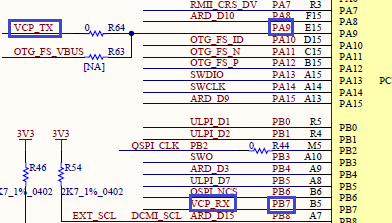


Let's look at the picture of the controller, which legs we automatically connected as USART generator Cube MX



These legs are PB7 as RX and PB6 as TX.

Let's check on the diagram, is this true?

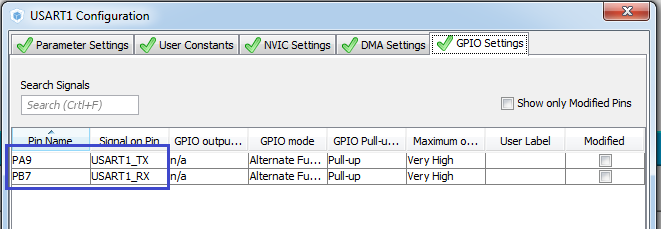


As we see, RX is the right one. and TX - no. Therefore, we redefine it to PA9.

And this is the situation very often. The same will happen when FMC and LTDC are turned on.

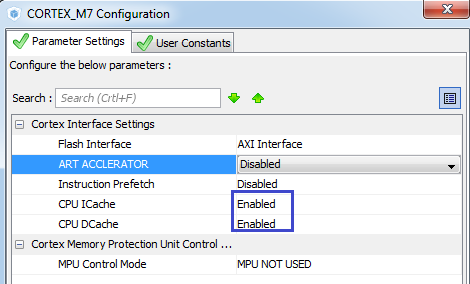
Well, most likely, Cube MX and do not have to know what kind of contacts we have and what is involved in the circuit of the debug card. We did not choose a board, but a controller. Many advise choosing a fee, but I do not like it. All yellow and green, there are no free legs at all and, conversely, you begin to think what to turn off. Therefore, it is better to look at the diagram sometimes.

In general with the USART figured out

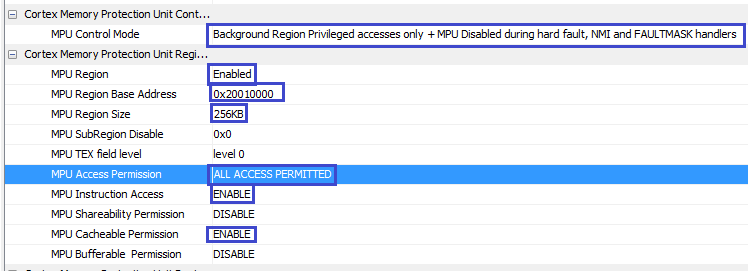


Now let's look at the controller itself.

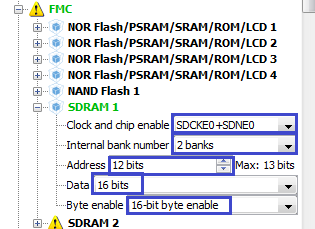
Let's go into Configuration and open there CORTEX M7. Turn on the first thing cache



Further access regions

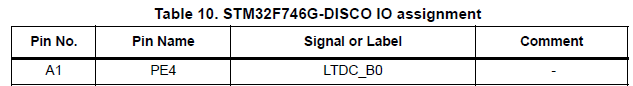


Now FMC.



We see that we have a lot of legs turned on and a lot of them will be redefined according to the scheme, so I'll write a list of what should be included the legs for FMC and in what capacity, and then how little, maybe to someone turn on at all others. We already know which legs are responsible for what, therefore it will be easier for us to understand the pinouts.

Also a clue is the table 10 in the User Manual

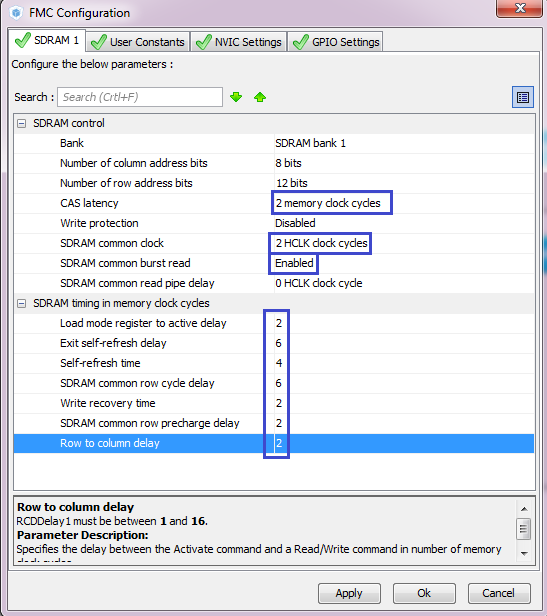


From it we all collect

| **Pin** | **Pin Name** | **Signal or Label** |
| --- | --- | --- |
| A5 | PE1 | FMC\_NBL1 |
| A6 | PE0 | FMC\_NBL0 |
| B7 | PG15 | FMC\_SDNCAS |
| B12 | PD0 | FMC\_D2 |
| C12 | PD1 | FMC\_D3 |
| D2 | PF0 | FMC\_A0 |
| E2 | PF1 | FMC\_A1 |
| G2 | PF2 | FMC\_A2 |
| H2 | PF3 | FMC\_A3 |
| H14 | PG8 | FMC\_SDCLK |
| J2 | PF4 | FMC\_A4 |
| J3 | PH5 | FMC\_SDNWE |
| J4 | PH3 | FMC\_SDNE0 |
| K3 | PF5 | FMC\_A5 |
| K13 | PD15 | FMC\_D1 |
| K15 | PD10 | FMC\_D15 |
| L4 | PC3 | FMC\_SDCKE0 |
| L12 | PD14 | FMC\_D0 |
| L14 | PD9 | FMC\_D14 |
| L15 | PD8 | FMC\_D13 |
| M6 | PF12 | FMC\_A6 |
| M7 | PG1 | FMC\_A11 |
| M8 | PF15 | FMC\_A9 |
| N6 | PF13 | FMC\_A7 |
| N7 | PG0 | FMC\_A10 |
| N9 | PE8 | FMC\_D5 |
| N12 | PG4 | FMC\_BA0 |
| P6 | PF14 | FMC\_A8 |
| P8 | PF11 | FMC\_SDNRAS |
| P9 | PE9 | FMC\_D6 |
| P10 | PE11 | FMC\_D8 |
| P11 | PE14 | FMC\_D11 |
| R8 | PE7 | FMC\_D4 |
| R9 | PE10 | FMC\_D7 |
| R10 | PE12 | FMC\_D9 |
| R11 | PE15 | FMC\_D12 |
| R12 | PE13 | FMC\_D10 |

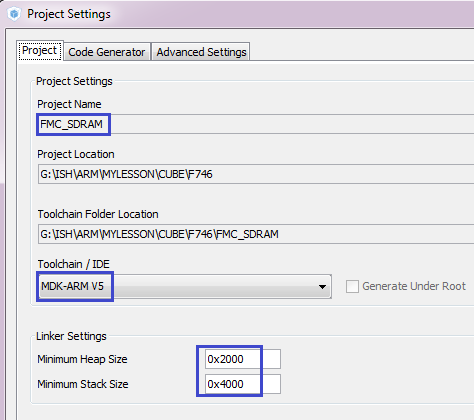
That's the way we redefine contacts.

Now configure our FMC in the Configuration tab



Here everything is according to the documentation. We added latency to the bus by enabling the column addressing, added one more clock to the commands, turned on the BRUST (packet data mode) for reading, and also set the timings.

We will configure the project, calling it FMC\_SDRAM, selecting Keil 5 as the environment, and also increase the stack and the heap by 10 times



In general, with the Cube MX so far.

Generate the project, open it in Keil, setting up the programmer for auto-cutting.

First, write a simple test in main () in an infinite loop to check the green LED

    / \* USER CODE BEGIN 3 \* /

**HAL\_GPIO\_WritePin (GPIOI, GPIO\_PIN\_1, GPIO\_PIN\_RESET);**

**HAL\_Delay (500);**

**HAL\_GPIO\_WritePin (GPIOI, GPIO\_PIN\_1, GPIO\_PIN\_SET);**

**HAL\_Delay (500);**

  }

  / \* USER CODE END 3 \* /

We will collect the code, we will sew the controller, the LED will flash.

Now check **USART**

Connect the file to main.c

/ \* USER CODE BEGIN Includes \* /

**#include "string.h"**

In main () we'll create a small string buffer

  / \* USER CODE BEGIN 1 \* /

**char str1 [20] = {0};**

  / \* USER CODE END 1 \* /

Let's add code to an infinite loop

  HAL\_Delay (500); }

**sprintf (str1, "Ok! rn");**

**HAL\_UART\_Transmit (& huart1, (uint8\_t \*) str1, strlen (str1), 0x1000);**

Run the terminal program, set it to 115200 with one stop bit and press **Connect** .

Then we'll compile the code, we'll tell the controller and look at the terminal

image19

Everything is fine.

Now let's take care of SDRAM memory.

Part of the memory was already initialized in the Cube MX.

Continue the initialization further.

Let's create two files **MT48LC4M32B2.c** and **MT48LC4M32B2.h in the** usual way

Connect the header file to main.c and to MT48LC4M32B2.c

/ \* USER CODE BEGIN Includes \* /

#include "MT48LC4M32B2.h"

#include "string.h"

#include "MT48LC4M32B2.h"

In this header file we will add macros for configuring and accessing **SDRAM memory** , taking them from the example, and also we will connect the **HAL** library there .

As a result, the file will be

**#ifndef \_\_MT48LC4M32B2\_H**

**#define \_\_MT48LC4M32B2\_H**

**// ----------------**

**#include "stm32f7xx\_hal.h"**

**// ----------------**

**#define SDRAM\_TIMEOUT ((uint32\_t) 0xFFFF)**

**#define SDRAM\_MODEREG\_BURST\_LENGTH\_1 ((uint16\_t) 0x0000)**

**#define SDRAM\_MODEREG\_BURST\_LENGTH\_2 ((uint16\_t) 0x0001)**

**#define SDRAM\_MODEREG\_BURST\_LENGTH\_4 ((uint16\_t) 0x0002)**

**#define SDRAM\_MODEREG\_BURST\_LENGTH\_8 ((uint16\_t) 0x0004)**

**#define SDRAM\_MODEREG\_BURST\_TYPE\_SEQUENTIAL ((uint16\_t) 0x0000)**

**#define SDRAM\_MODEREG\_BURST\_TYPE\_INTERLEAVED ((uint16\_t) 0x0008)**

**#define SDRAM\_MODEREG\_CAS\_LATENCY\_2 ((uint16\_t) 0x0020)**

**#define SDRAM\_MODEREG\_CAS\_LATENCY\_3 ((uint16\_t) 0x0030)**

**#define SDRAM\_MODEREG\_OPERATING\_MODE\_STANDARD ((uint16\_t) 0x0000)**

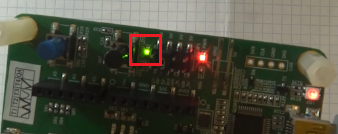
**#define SDRAM\_MODEREG\_WRITEBURST\_MODE\_PROGRAMMED ((uint16\_t) 0x0000)**

**#define SDRAM\_MODEREG\_WRITEBURST\_MODE\_SINGLE ((uint16\_t) 0x0200)**

**// ----------------**

**#endif / \* \_\_MT48LC4M32B2\_H \* /**

We will collect the project, we will impose the controller and check if the LED is flashing, that is, whether we use the port's foot



The LED at us blinks as it should be once a second.

We will postpone further work with the project until the [**next part of**](http://narodstream.ru/stm-urok-62-fmc-sdram-chast-3/) our lesson

**Lesson 62**

**Part 3**

# FMC SDRAM

In the [**last part of**](http://narodstream.ru/stm-urok-62-fmc-sdram-chast-2/) our lesson, we set up our FMC controller, generated and configured the project, and also checked the design of the project for the flashing of the LED.

Now start the initialization of our FMC, since any peripherals in this necessarily need.

Add the initialization function to the file MT48LC4M32B2.c

**void MT48LC4M32B2\_Init(SDRAM\_HandleTypeDef \*hsdram)**

**{**

**}**

Let's make it a prototype and call it in main ()

  / \* USER CODE BEGIN 2 \* /

**MT48LC4M32B2\_Init (& hsdram1);**

Also in the file MT48LC4M32B2.c add a global variable for the FMC command one more for the return status from the HAL functions

#include "MT48LC4M32B2.h"

**FMC\_SDRAM\_CommandTypeDef command;**

**HAL\_StatusTypeDef hal\_stat;**

Now you can start writing the initialization function.

Create an unoptimized local variable there and send the first command to the FMC

void MT48LC4M32B2\_Init (SDRAM\_HandleTypeDef \* hsdram)

{

**\_\_IO uint32\_t tmpmrd = 0;**

**command.CommandMode = FMC\_SDRAM\_CMD\_CLK\_ENABLE;**

**command.CommandTarget = FMC\_SDRAM\_CMD\_TARGET\_BANK1;**

**command.AutoRefreshNumber = 1;**

**command.ModeRegisterDefinition = 0;**

**hal\_stat = HAL\_SDRAM\_SendCommand (hsdram, & command, SDRAM\_TIMEOUT);**

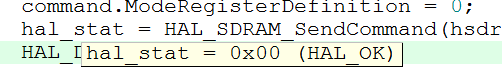
**HAL\_Delay (1);**

}

The delay is needed for the command to run, and it will also be convenient for us to have the code execution pointer appear during debugging to see the status.

In this command, we include timing, addressing exactly 1 memory bank, one auto-update (sometimes from 1 to 16). In the registers of registers, we do not enter anything.

Let's put a breakpoint for this delay, compile the code, go into debugging, run for execution and after stopping at this point, let's look at the status



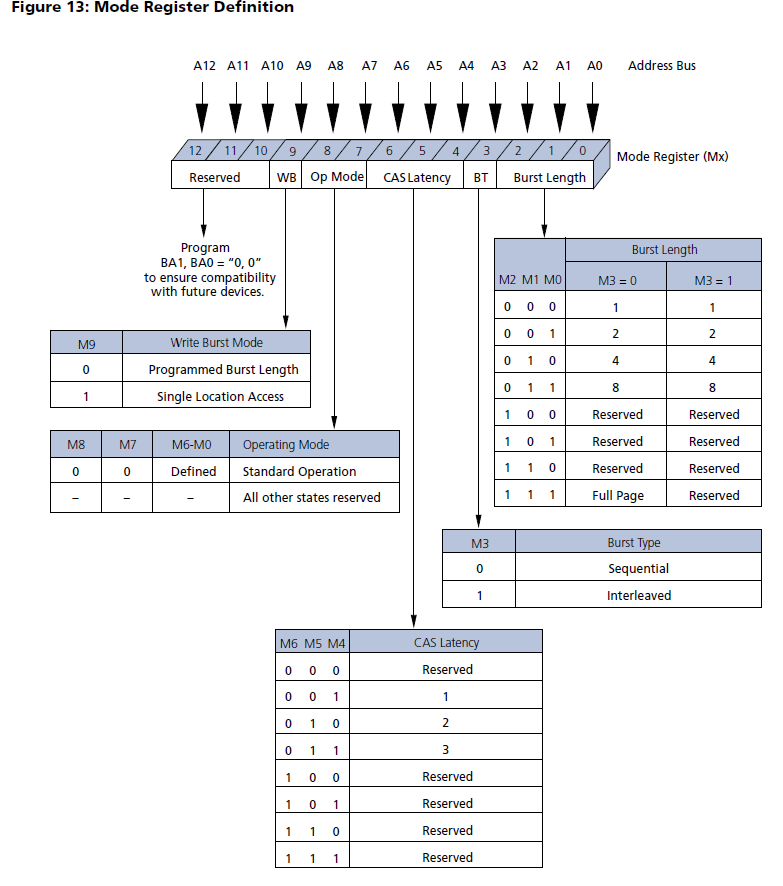
Everything is normal!

Let's look at this register of modes. He knows a lot, of course, we will not analyze all his possibilities in this lesson, since they do not need us. We have the task of counting FMC on making memory flat so that it can be conveniently accessed for writing and reading

Defines the number of consecutive auto refresh command issued

                                              in auto refresh mode.

                                              This parameter can be a value between Min\_Data = 1 and Max\_Data = 16



This register is able to store certain settings, looking at which, SDRAM behaves in certain conditions behaves in one way or another.

Here, the mode and type of BRUST, the packet length, latency, operation mode are adjusted.

Here such here, in general, the register. We will return to him.

Send the following command in the initialization function

  HAL\_Delay (1);

**command.CommandMode = FMC\_SDRAM\_CMD\_PALL;**

**command.CommandTarget = FMC\_SDRAM\_CMD\_TARGET\_BANK1;**

**command.AutoRefreshNumber = 1;**

**command.ModeRegisterDefinition = 0;**

**hal\_stat = HAL\_SDRAM\_SendCommand (hsdram, & command, SDRAM\_TIMEOUT);**

**HAL\_Delay (1);**

There is no need for delay here, and we use it only to see the status of the return.

Here we send another command FMC\_SDRAM\_CMD\_PALL, which deactivates all memory banks before the regeneration (PALL - precharged all).

Also we will collect the code and check the status in debugging.

Since we will have many more teams, we will do this thing in the [**next part of**](http://narodstream.ru/stm-urok-62-fmc-sdram-chast-4/) our lesson.

**Lesson 62**

**Part 4**

# FMC SDRAM

In the [**last part of the**](http://narodstream.ru/stm-urok-62-fmc-sdram-chast-3/)  lesson, we learned how to send commands to the **SDRAM** memory chip via the **FMC** controller , using only the functions of the **HAL** library .

Also we already passed two teams, and, moreover, checked that they were successfully transferred.

The following command

  HAL\_Delay (1);

**command.CommandMode = FMC\_SDRAM\_CMD\_AUTOREFRESH\_MODE;**

**command.CommandTarget = FMC\_SDRAM\_CMD\_TARGET\_BANK1;**

**command.AutoRefreshNumber = 8;**

**command.ModeRegisterDefinition = 0;**

**hal\_stat = HAL\_SDRAM\_SendCommand (hsdram, & command, SDRAM\_TIMEOUT);**

**HAL\_Delay (1);**

Here we set the auto-regeneration mode necessary for the normal operation of the SDRAM.

The following command

**tmpmrd = (uint32\_t) SDRAM\_MODEREG\_BURST\_LENGTH\_1 |**

**SDRAM\_MODEREG\_BURST\_TYPE\_SEQUENTIAL |**

**SDRAM\_MODEREG\_CAS\_LATENCY\_2 |**

**SDRAM\_MODEREG\_OPERATING\_MODE\_STANDARD |**

**SDRAM\_MODEREG\_WRITEBURST\_MODE\_SINGLE;**

**command.CommandMode = FMC\_SDRAM\_CMD\_LOAD\_MODE;**

**command.CommandTarget = FMC\_SDRAM\_CMD\_TARGET\_BANK1;**

**command.AutoRefreshNumber = 1;**

**command.ModeRegisterDefinition = tmpmrd;**

**hal\_stat = HAL\_SDRAM\_SendCommand (hsdram, & command, SDRAM\_TIMEOUT);**

Here we include the packet size, sequential transmission mode, latency 2 cycles, standard mode, single mode of packet recording.

And at the end of the initialization we manually **enter** into the register of the FMC **SDRTR** , responsible for the time of auto-regeneration send a certain number (explained in the comment to the code)

**/ \* Step 8: Set the refresh rate counter \* /**

**/ \* (15.62 us x Freq) - 20 \* /**

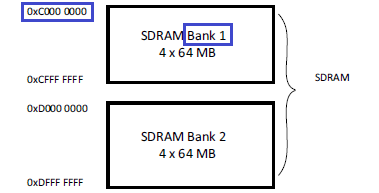
**/ \* Set the device refresh counter \* /**

**hsdram-> Instance-> SDRTR | = ((uint32\_t) ((1292) << 1));**

}

Here. " actually the whole initialization.

Also we know that FMC displays addresses of SDRAM at certain addresses. Let's open page 307 in the Reference Manual on the controller



Since we connected exactly Bank 1, we will write another macro to the file MT48LC4M32B2.h, so that it would be more convenient and more convenient to access the SDRAM from the program

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

**#define SDRAM\_BANK\_ADDR ((uint32\_t) 0xC0000000)**

And also write a few macros and global variables in the main.c file

/ \* USER CODE BEGIN PV \* /

/ \* Private variables ------------------- \* /

**#define BUFFER\_SIZE ((uint32\_t) 0x0100)**

**#define WRITE\_READ\_ADDR ((uint32\_t) 0x0800)**

**uint32\_t aTxBuffer [BUFFER\_SIZE];**

**uint32\_t aRxBuffer [BUFFER\_SIZE];**

**uint32\_t uwIndex = 0;**

/ \* USER CODE END PV \* /

Here are the macros for the buffer size, that is, the number of bytes that we will write to the SDRAM and then read from there, the offset from the start of the SDRAM to the FMC, the write buffer, the read buffer, and the sequence number in the array.

Also add a function to the main.c file to fill the buffer with certain data for later use for writing to SDRAM. The function was taken from the example of the Cube MX repository and does not need changes

/ \* USER CODE BEGIN 0 \* /

**static void Fill\_Buffer (uint32\_t \* pBuffer, uint32\_t uwBufferLenght, uint32\_t uwOffset)**

**{**

**uint32\_t tmpIndex = 0;**

**/ \* Put in global buffer different values ​​\* /**

**for (tmpIndex = 0; tmpIndex <uwBufferLenght; tmpIndex ++)**

**{**

**pBuffer [tmpIndex] = tmpIndex + uwOffset;**

**}**

**}**

/ \* USER CODE END 0 \* /

In the incoming parameters, we have a pointer to the buffer that we will fill with data, the number of bytes to fill, and the value of the initial byte. Further byte values ​​will increase by 1.

Using this function and calling it in main (), fill our buffer, starting with any value

 / \* USER CODE BEGIN 2 \* /

 MT48LC4M32B2\_Init (& hsdram1);

**Fill\_Buffer (aTxBuffer, BUFFER\_SIZE, 0x37BA0F68);**

Now write to SDRAM the contents of the buffer

 Fill\_Buffer (aTxBuffer, BUFFER\_SIZE, 0x37BA0F68);

**// write the data from the buffer to the SDRAM from address 0xC0000800**

**for (uwIndex = 0; uwIndex <BUFFER\_SIZE; uwIndex ++)**

**{**

**\* (\_\_ IO uint32\_t \*) (SDRAM\_BANK\_ADDR + WRITE\_READ\_ADDR + 4 \* uwIndex) = aTxBuffer [uwIndex];**

**}**

After that, try to read them from SDRAM to another buffer

  }

**// read the data from the SDRAM from the address 0xC0000800 to another buffer**

**for (uwIndex = 0; uwIndex <BUFFER\_SIZE; uwIndex ++)**

**{**

**aRxBuffer [uwIndex] = \* (\_\_ IO uint32\_t \*) (SDRAM\_BANK\_ADDR + WRITE\_READ\_ADDR + 4 \* uwIndex);**

**}**

Then we will send the bytes in text form in USART

  }

**// send the bytes to USART**

**for (uwIndex = 0; uwIndex <BUFFER\_SIZE; uwIndex ++)**

**{**

**sprintf (str1, "% 03ld: 0x% 08lXrn", (unsigned long) uwIndex, (unsigned long) aRxBuffer [uwIndex]);**

**HAL\_UART\_Transmit (& huart1, (uint8\_t \*) str1, strlen (str1), 0x1000);**

**HAL\_Delay (100);**

**}**

The delay here is optimal, since if less, the terminal program hangs, others do not hang, but they are not so intuitive.

At the end of the transfer, let's light the green LED

  }

**HAL\_GPIO\_WritePin (GPIOI, GPIO\_PIN\_1, GPIO\_PIN\_SET);**

From an infinite loop, everything is deleted

  while (1)

  {

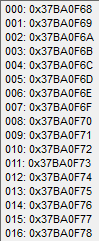
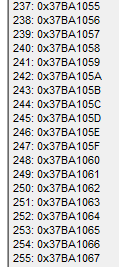
  / \* USER CODE END WHILE \* /

  / \* USER CODE BEGIN 3 \* /

 }

We'll collect the code, open the terminal program, click Connect, select the port and enter the controller.

We should see the following picture in the terminal

Well! We all counted from SDRAM memory, since all bytes came in USART.

I think we will return to the topic of FMC and SDRAM when we program the display, as well as in other cases.