1.9.3 Case study: Using the STM32F Flexible Memory Controller to access SDRAM

The Flexible Memory Controller (FMC) found in STM32 microcontrollers consists of the following main blocks:

1. the interface to the CPU's Advanced High-performance Bus (AHB),
2. the NOR Flash/SRAM memory controller,
3. the SDRAM memory controller, and
4. NAND Flash controller.

The block diagram of the FMC is shown in Figure 1.32. The AHB interface allows the CPU (and other bus master peripherals) to access the external memories through the FMC controller. Two primary purposes of FMC are to translate transactions on the high-speed CPU bus (namely AHB bus) into the appropriate external protocol and to meet the access time requirements of the external memory devices.

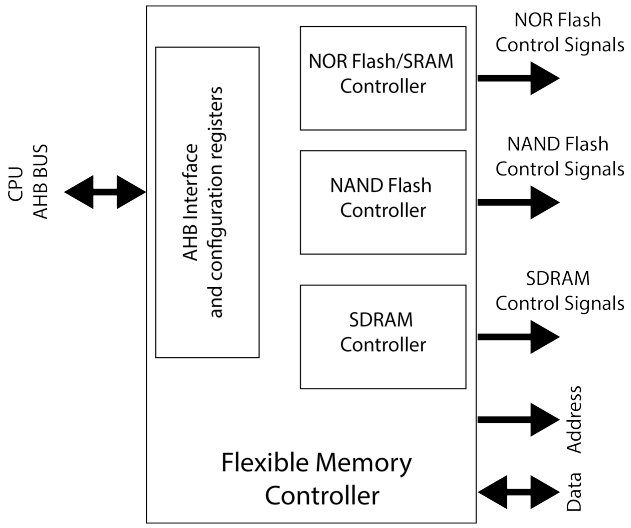

Fig. 1.32: FMC block diagram.

From the FMC (or microprocessor) point of view, the external memory is divided into six fixed-size regions of 256 Mbytes each, called banks (Figure 1.33). The first bank is used to address NOR Flash memory devices. The third bank is used to address NAND Flash memory devices. The last two banks are used to address two SDRAM devices (one device per bank). The address bit 28 on the AHB bus (internal AHB address line 28) selects one of the memory devices (banks). Let us focus only on the FMC SDRAM controller and an SDRAM device in the fifth bank.

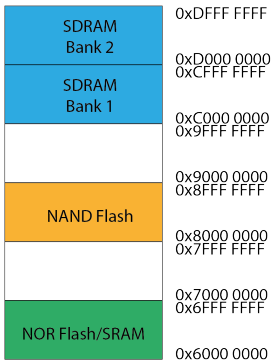

Fig. 1.33: Memory regions accessible from the FMC controller.

All external memories share the addresses, data and control signals with the controller, and each external device is accessed utilizing a unique chip-select signal. The FMC performs only one access at a time to an external device. Here, we will describe only the SDRAM controller and its use to interface a 128 Mbit SDRAM memory chip. All AHB transactions, in this case, translate into the SDRAM device protocol.

The FMC SDRAM controller supports SDRAM devices of up to 256 Mbytes. It can issue a 13-bit row address, an 11-bit column address, and a 2-bit bank address. The memory accesses can be 8-bit, 16-bit, and 32-bit. We will use Micron's 1 Meg x 32 x 4 banks MT48LC4M32B2 SDRAM chip, organized as 4096 rows x 256 columns x 32 bits per bank. Hence, the memory controller would issue a 12-bit row address, an 8-bit column address, and a 2-bit bank address.

The SDRAM controller in Figure 1.34 accepts single and burst read and write requests and translates them into single memory accesses. In both cases, the SDRAM controller keeps track of the active row in each bank to be able to perform consecutive read and write accesses. The FMC SDRAM controller comprises a read FIFO (6 lines x 32 bits). It is used to read data in advance - the memory controller anticipates READ commands to the open row if the RBURST bit is set in the FMC_SDCRx register and stores data in the FIFO. Two bits RPIPE[1:0] in the FMC_SDCRx register defines how much data will be anticipated and stored into the FIFO during the read access. If we set both RPIPE[1:0] bits to zero, four data will be anticipated during a single read access. The first read data will be transmitted to the AHB bus, and the other three will be stored in the read FIFO buffer. The read FIFO buffer stores a 14-bit address tag for each line to identify its content: 11 bits for the column address, 2 bits for the internal bank in the active row, and 1 bit for the SDRAM device. Each time a read request occurs, the SDRAM controller checks if the address matches one of the address tags in the read FIFO buffer. In such a case, data are directly read from the FIFO buffer. Otherwise, a new read command is issued to the SDRAM device, and new data is read to the FIFO buffer.

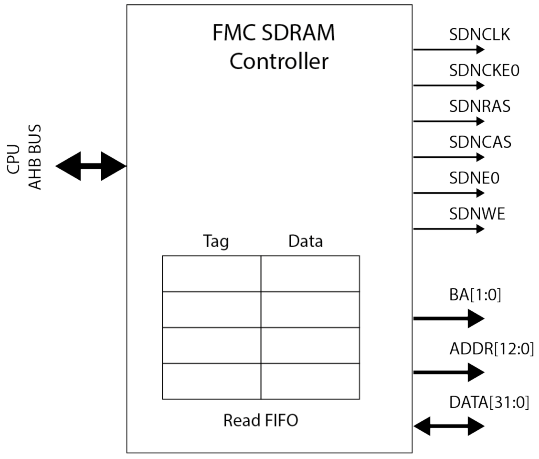

Fig. 1.34: FMC SDRAM Controller block diagram and signals.

The FMC SDRAM controller periodically issues auto-refresh commands to refresh the SDRAM. The programmer should initialize the internal counter value in the FMC_SDRTR. This value defines the number of memory clock cycles between two refresh cycles (refresh rate). When this counter reaches zero, the FMC SDRAM controller issues the auto-refresh command. If there is an ongoing memory access, the auto-refresh request is delayed until the memory access finishes; otherwise, the auto-refresh request takes precedence. If the memory access request occurs during an auto-refresh operation, the request is buffered and processed when the auto-refresh completes.

For our particular case, where the FMC SDRAM controller is used to access the MT48LC4M32B2 SDRAM chip, the 32-bit memory address from the AHB bus is mapped into the SDRAM address as presented in Figure 1.35. This figure illustrates how the 32-bit addresses issued by the CPU on the AHB bus map to the 26-bit addresses issued by the SDRAM controller to the SDRAM device.

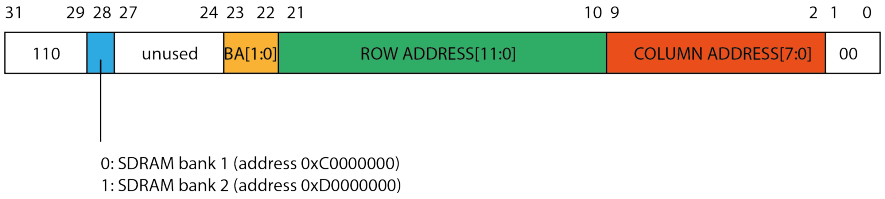

Fig. 1.35: Address maping for a 128-bit SDRAM (4096 rows x 256 columns x 4 banks x 32 bit).

In order to use the FMC SDRAM controller with an external SDRAM device residing in the SDRAM Bank 1, we should:

1. first, initialize the FMC SDRAM controller according to the used SDRAM device, and
2. secondly, initialize the SDRAM device.

The first step involves programming two FMC SDRAM controller configuration registers, SDRAM Control Register 1 (FMC_SDCR1) and SDRAM Timing Register 1 (FMC_SDTR1). The bits in FMC_SDCR1 (Figure 1.37) define the SDRAM clock period, CAS Latency, whether the FMC anticipates READ commands (burst read), data bus width and the internal organization of the SDRAM chip (rows, columns and banks).

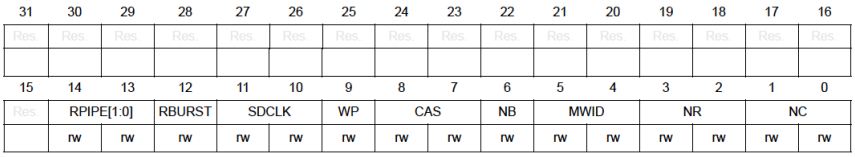

Fig. 1.36: Control register (FMC_SDCR).

The bits in FMC_SDTR1 define SDRAM timing parameters, e.g. RAS-to-CAS delay, row-precharge delay, etc. In order to correctly set the bits in these two registers, we should consult the datasheet for a particular SDRAM chip.

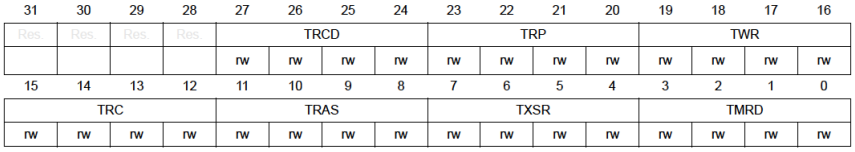

Fig. 1.37: Timing register (FMC_SDTR).

The second step initializes the SDRAM chip. During the SDRAM chip initialization, the FMC controller sends several predefined commands to the SDRAM chip. To send these commands, we should write them into the FMC SDRAM Command Mode Register (FMC_SDCMR). The required initialization steps are described in the datasheet for a particular SDRAM chip and involve the following:

1. providing stable CLOCK signal,
2. performing a PRECHARGE ALL command, which puts all rows in all banks into an idle state,

3. issuing several AUTO REFRESH commands
4. issuing several NOP commands before SDRAM is ready for access.

Instead of directly setting bits in the FMC SDRAM configuration registers, we will rather use the HAL library. The HAL library abstracts most of the FMC SDRAM controller hardware details. The FMC SDRAM controller is abstracted in HAL with the `SDRAM_HandleTypeDef` C structure. The two most important members of this structure are the C reference to `FMC_SDRAM_TypeDef Instance` structure and `FMC_SDRAM_InitTypeDef Init`. The `Instance` is a reference to the SDRAM registers (it holds the base address of the FMC SDRAM registers), while the `Init` structure allows for FMC SDRAM controller configuration. The `FMC_SDRAM_InitTypeDef Init` structure is defined as follows:

```
typedef struct
{
  uint32_t SDBank;
  uint32_t ColumnBitsNumber;
  uint32_t RowBitsNumber;
  uint32_t MemoryDataWidth;
  uint32_t InternalBankNumber;
  uint32_t CASLatency;
  uint32_t WriteProtection;
  uint32_t SDClockPeriod;
  uint32_t ReadBurst;
} FMC_SDRAM_InitTypeDef;
```

Listing 1.1: FMC SDRAM `FMC_SDRAM_InitTypeDef` C structure.

Let us briefly describe the elements of the `FMC_SDRAM_InitTypeDef Init` structure:

- `SDBank`: Specifies the SDRAM memory device that will be used (bank 1 or bank 2 according to Figure 1.33).
- `ColumnBitsNumber`: Defines the number of bits of the column address.
- `RowBitsNumber`: Defines the number of bits of the row address.
- `MemoryDataWidth`: Defines the memory device width.
- `InternalBankNumber`: Defines the number of the device's internal banks.
- `CASLatency`: Defines the SDRAM CAS latency in the number of memory clock cycles.
- `WriteProtection`: Enables/Disables the SDRAM device to be accessed in write mode.
- `SDClockPeriod`: Defines the SDRAM Clock Period for SDRAM devices. The SDRAM clock period can be HCLK/2 or HCLK/3, where HCLK is the clock period on the CPU's AHB bus.
- `ReadBurst`: Enables the SDRAM controller to anticipate the next read commands during the CAS latency and stores data in the Read FIFO.

Besides the `FMC_SDRAM_InitTypeDef` C structure, which abstracts the content of FMC_SDCR1 register, the `FMC_SDRAM_TimingTypeDef` C structure is used to abstract the content of FMC_SDTR1 register. It is defined as follows:

```
typedef struct
{
  uint32_t LoadToActiveDelay;
  uint32_t ExitSelfRefreshDelay;
  uint32_t SelfRefreshTime;
  uint32_t RowCycleDelay;
  uint32_t WriteRecoveryTime;
  uint32_t RPDelay;
  uint32_t RCDDelay;
} FMC_SDRAM_TimingTypeDef;
```

Listing 1.2: FMC SDRAM `FMC_SDRAM_TimingTypeDef` C structure.

The elements of the `FMC_SDRAM_TimingTypeDef` C structure are self-explanatory (they represent the particular timings for SDRAM chips), and there is no need to describe them.

The code Listing 1.3 shows the FMC SDRAM controller initialization.

```
uint8_t Init_SDRAM(void)
{
  static uint8_t sdramstatus = SDRAM_ERROR;
  /* SDRAM device configuration */
  sdramHand.Instance = FMC_SDRAM_DEVICE;

  /* Timing configuration for 100Mhz as SDRAM clock frequency
     (System clock is up to 200Mhz) */
  /* These parameters are from the MT48LC4M32B2 Data Sheet,
     Table 18 and Table 19 */
  sdramTiming.LoadToActiveDelay    = 2;   // t_MRD
  sdramTiming.ExitSelfRefreshDelay = 7;   // t_XSR
  sdramTiming.SelfRefreshTime      = 5;   // t_RAS
  sdramTiming.RowCycleDelay        = 7;   // t_RC
  sdramTiming.WriteRecoveryTime    = 2;   // t_WR
  sdramTiming.RPDelay              = 2;   // t_RP
  sdramTiming.RCDDelay             = 2;   // t_RCD


  sdramHand.Init.SDBank             = FMC_SDRAM_BANK1;
  sdramHand.Init.ColumnBitsNumber   = FMC_SDRAM_COLUMN_BITS_NUM_8;
  sdramHand.Init.RowBitsNumber      = FMC_SDRAM_ROW_BITS_NUM_12;
  sdramHand.Init.MemoryDataWidth    = FMC_SDRAM_MEM_BUS_WIDTH_32;
  sdramHand.Init.InternalBankNumber = FMC_SDRAM_INTERN_BANKS_NUM_4;
  sdramHand.Init.CASLatency         = FMC_SDRAM_CAS_LATENCY_3;
  sdramHand.Init.WriteProtection  = FMC_SDRAM_WRITE_PROTECTION_DISABLE;
  sdramHand.Init.SDClockPeriod      = FMC_SDRAM_CLOCK_PERIOD_2;
  sdramHand.Init.ReadBurst          = FMC_SDRAM_RBURST_ENABLE;
  sdramHand.Init.ReadPipeDelay      = FMC_SDRAM_RPIPE_DELAY_0;

  /* SDRAM controller initialization */

  if(HAL_SDRAM_Init(&sdramHand, &sdramTiming) != HAL_OK)
  {
    sdramstatus = SDRAM_ERROR;
  }
  else
  {
    sdramstatus = SDRAM_OK;
  }

  /* Once the FMC SDRAM Ctrl is initialized, we can access
     and initialize the SDRAM chip */
  /* SDRAM initialization sequence */
  SDRAM_Initialization_sequence(REFRESH_COUNT);
```

```
46
  return sdramstatus;
48 }
```

Listing 1.3: FMC SDRAM Controller initialization.

Firstly, we set the SDRAM timing parameters (in the FMC_SDTR1 register) considering the 100MHz SDRAM clock, then we set the SDRAM configuration (in the FMC_SDCR1 register). To initialize the FMC SDRAM controller (that is to copy the elements of both C structures into the appropriate fields of the FMC_SDCR1 and FMC_SDTR1 registers), we call the HAL function `HAL_SDRAM_Init(SDRAM_HandleTypeDef *hsdram, FMC_SDRAM_TimingTypeDef *Timing)`.

After the FMC SDRAM initialization, we should initialize the SDRAM chip. SDRAMs must be powered up and initialized in a predefined manner. This is a necessary step required to put all SDRAM rows in the idle state (precharge all rows) and prepare the SDRAM chip for accepting and executing the commands. The SDRAM initialization sequence is described in the SDRAM datasheet in detail. The code Listing 1.4 shows the FMC SDRAM chip initialization. Briefly, the initialization procedure contains four steps:

1. Enable the stable SDRAM clock.
2. Wait for at least 100us prior to issuing any command.
3. Perform a PRECHARGE ALL command.
4. Issue at least two AUTO REFRESH commands.
5. The SDRAM is now ready for mode register programming. Because the mode register will power up in an unknown state, it should be loaded with desired bit values prior to applying any operational command.

```
/**
2   * @brief  Init the SDRAM device.
  * SDRAMs must be initialized in a predefined manner. Operational ↩
    procedures
4   * other than those specified in the SDRAM Data Sheet may result in ↩
    undefined operation.
  * @param  RefreshCount: SDRAM refresh counter value
6   * @retval None
  */
8 void SDRAM_Initialization_sequence(uint32_t RefreshCount)
{
10   __IO uint32_t tmpmrd = 0;

12   /* Step 1: Configure a clock configuration enable command */
  sdramCmd.CommandMode            = FMC_SDRAM_CMD_CLK_ENABLE;
14   sdramCmd.CommandTarget          = FMC_SDRAM_CMD_TARGET_BANK1;
  sdramCmd.AutoRefreshNumber      = 1;
16   sdramCmd.ModeRegisterDefinition = 0;

18

  /* Send the Clock Configuration Enable command to the target bank*/
20   /* The command is sent as soon as the Command MODE field in the
    CMR is written */
22   HAL_SDRAM_SendCommand(&sdramHand, &sdramCmd, SDRAM_TIMEOUT);

24   /*
```

```
    * Once the clock is stable, the SDRAM requires a 100us delay
    * prior to issuing any command
    */

  /* Step 2: Insert 100 us minimum delay */
  /* Inserted delay is equal to 1 ms due to systick time base unit */
  HAL_Delay(1);


  /*
   * Once the 100us delay has been satisfied, a PRECHARGE command
   * should be applied. All banks must then be precharged,
   * thereby placing the device in the all banks idle state.
   */
  /* Step 3: Configure a PALL (precharge all) command */
  sdramCmd.CommandMode            = FMC_SDRAM_CMD_PALL;
  sdramCmd.CommandTarget          = FMC_SDRAM_CMD_TARGET_BANK1;
  sdramCmd.AutoRefreshNumber      = 1;
  sdramCmd.ModeRegisterDefinition = 0;

  /* Send the Precharge All command to the target bank */
  /* The command is sent as soon as the Command MODE field
     in the CMR is written */
  HAL_SDRAM_SendCommand(&sdramHand, &sdramCmd, SDRAM_TIMEOUT);

  /*
   * Once in the idle state, at least two AUTO REFRESH cycles must
   * be performed. If desired, more than two AUTO REFRESH
   * commands can be issued in the sequence.
   */
  /* Step 4: Configure an Auto Refresh command */
  sdramCmd.CommandMode            = FMC_SDRAM_CMD_AUTOREFRESH_MODE;
  sdramCmd.CommandTarget          = FMC_SDRAM_CMD_TARGET_BANK1;
  sdramCmd.AutoRefreshNumber      = 8;
  sdramCmd.ModeRegisterDefinition = 0;

  /* Send the Auto-refresh commands to the target bank */
  /* The command is sent as soon as the Command MODE
     field in the CMR is written */
  HAL_SDRAM_SendCommand(&sdramHand, &sdramCmd, SDRAM_TIMEOUT);


  /*
   * The SDRAM is now ready for mode register programming.
   * Because the mode register will power up in an unknown state,
   * it should be loaded with desired bit values prior to
   * applying any operational command. Using the LMR command,
   * program the mode register.
   */
  /* Step 5: Program the external memory mode register */
  tmpmrd = (uint32_t)SDRAM_MODEREG_BURST_LENGTH_1          |\
                     SDRAM_MODEREG_BURST_TYPE_SEQUENTIAL   |\
                     SDRAM_MODEREG_CAS_LATENCY_3           |\
                     SDRAM_MODEREG_OPERATING_MODE_STANDARD |\
                     SDRAM_MODEREG_WRITEBURST_MODE_SINGLE;

  sdramCmd.CommandMode            = FMC_SDRAM_CMD_LOAD_MODE;
  sdramCmd.CommandTarget          = FMC_SDRAM_CMD_TARGET_BANK1;
  sdramCmd.AutoRefreshNumber      = 1;
  sdramCmd.ModeRegisterDefinition = tmpmrd;

  /* Send the Load Mode Register command to the target bank */
  /* The command is sent as soon as the Command MODE field in
     the CMR is written */
  HAL_SDRAM_SendCommand(&sdramHand, &sdramCmd, SDRAM_TIMEOUT);

  /*
```

```
92      * Wait for at least tMRD time. This is automatically performed by
        * the FMC SDRAM controller. At this point the DRAM is ready for
94      * any valid command.
        */
96
    /* Step 6: Set the refresh rate counter in Refresh Timer register */
98  /* This 13-bit field defines the refresh rate of the SDRAM device.
       It is expressed in number of memory clock cycles. */
100 HAL_SDRAM_ProgramRefreshRate(&sdramHand, RefreshCount);
}
```

Listing 1.4: SDRAM initialization sequence.

To enable the above procedure, the FMC SDRAM controller provides a special register called Command Mode register (FMC_SDCMR), illustrated in Figure 1.38. It contains four fields. The MODE field defines the command issued to

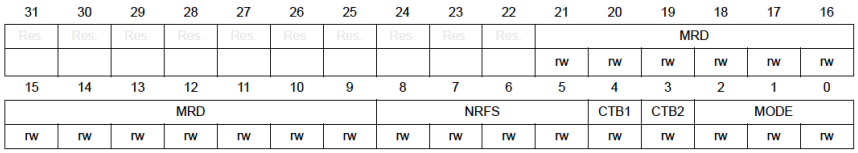

Fig. 1.38: Command Mode register (FMC_SDCMR).

the SDRAM chip. The possible commands are, for example, "CLK ENABLE", "PRECHARGE ALL", "AUTO REFRESH", and "LOAD MODE REGISTER". The CTB1 and CTB2 fields select the SDRAM chip to which the command is sent. As soon as the MODE field is written, the FMC SDRAM controller will issue the corresponding command to SDRAM chips selected by CTB1 and CTB2 command bits. The NRFS field defines how many consecutive Auto-refresh commands are issued in the fourth step of the initialization sequence, the MRD field contains the content that should be written to the SDRAM Mode Register. The mode register is a 12-bit special register inside the SDRAM chip and is used to define the specific mode of operation of the SDRAM. This definition includes the selection of a burst length (BL), a burst type, a CAS latency (CL), an operating mode and a write burst mode, as shown in Figure 1.39. The mode register is programmed from the FMC SDRAM controller via the "LOAD MODE REGISTER" command and retains the stored information until it is programmed again or the SDRAM device loses power.

The initialization of the SDRAM device is performed by sending a series of commands from the FMC_SDCMR register to the SDRAM device. Each command contains the actual instruction and its parameters. To facilitate the SDRAM chip initialization, HAL provides the `FMC_SDRAM_CommandTypeDef` C structure and `HAL_SDRAM_SendCommand` function. The `FMC_SDRAM_CommandTypeDef` C structure abstracts the content of FMC_SDCMR register and is defined as follows:

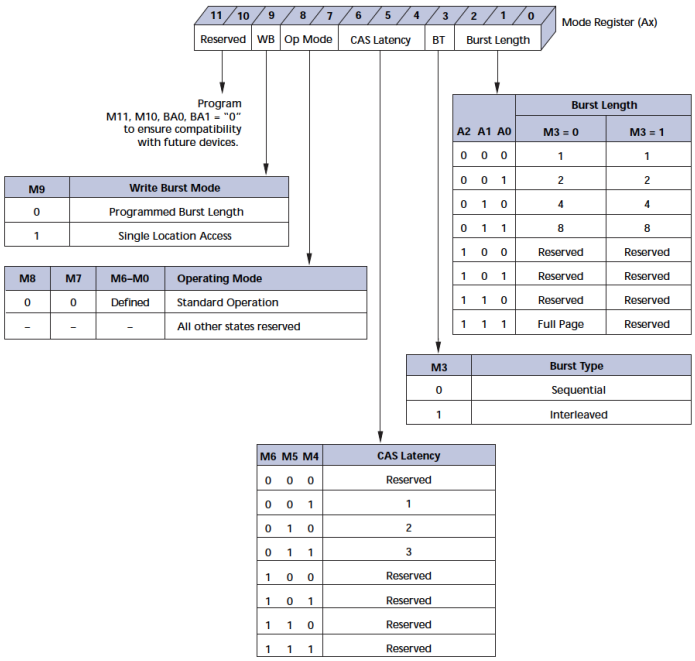

Fig. 1.39: SDRAM Mode Register.

```
typedef struct
{
  uint32_t CommandMode;
  uint32_t CommandTarget;
  uint32_t AutoRefreshNumber;
  uint32_t ModeRegisterDefinition;
} FMC_SDRAM_CommandTypeDef;
```

Listing 1.5: FMC SDRAM `FMC_SDRAM_CommandTypeDef` C structure.

Let us briefly describe the elements of the `FMC_SDRAM_CommandTypeDef Init` structure:

- `CommandMode`: Defines the command issued to the SDRAM device.
- `CommandTarget`: Defines which SDRAM device (1 or 2) the command will be issued to.
- `AutoRefreshNumber`: Defines the number of consecutive auto-refresh commands issued in auto-refresh mode.
- `ModeRegisterDefinition`: Defines the SDRAM Mode register content.

In order to send a command to the SDRAM device, we first fill the fields in the `FMC_SDRAM_CommandTypeDef Init` structure and then call the `HAL_SDRAM_SendCommand` function.

At the end of the SDRAM chip initialization, we set the auto-refresh period in the FMC SDRAM controller. The AUTO REFRESH command is used during the regular operation of the SDRAM to refresh its content. This command is nonpersistent, so it must be issued each time a refresh is required. If memory access is in progress, the auto-refresh request is delayed. The refresh controller inside the SDRAM chip generates the address of the row that should be refreshed. For example, the 128Mb SDRAM requires 4096 AUTO REFRESH commands every 64ms. To ensure that each row is refreshed according to this requirement, the SDRAM controller must issue an AUTO REFRESH command every 15.625us. The FMC SDRAM controller provides the Refresh Timer register (FMC_SDRTR). This register holds the 13-bit refresh rate in number of SDRAM clock cycles. This 13-bit field should be set immediately after the initialization of SDRAM. The 13-bit refresh rate is calculated as follows. As the SDRAM clock runs at 100 Mhz (10 ns period), 15.625 us equals 1562 SDRAM clock periods. We should subtract at least 20 SDRAM clock periods from this value to obtain a safe margin if an auto-refresh request occurs when a read request has been accepted. Hence, the 13-bit refresh rate in the FMC_SDRTR register corresponds to 1542.

To demonstrate the different scenarios when using the FMC SDRAM controller, we copy a matrix of size 64 rows times 256 columns from the external SDRAM to the internal SRAM. The elements of the matrix are 32-bit unsigned integers. In the first scenario (Listing 1.6), the matrix is accessed in row-major order, while in the second scenario (Listing 1.7), the matrix is accessed in column-major order. The constants `PA3_SDRAM_DEVICE_ADDR` and `SDRAM_COLS` in Listings 1.6 and 1.7 equal 0xC0008000 and 256, respectively. Hence, the matrix is read from the SDRAM startin at address 0xC000800.

```
void SDRAM_mat_row_access_test(void){
  volatile uint32_t address;

  for (int i = 0; i<MAT_ROWS; i++) {
    for(int j=0; j<SDRAM_COLS; j++) {
      address = PA3_SDRAM_DEVICE_ADDR + ((i*SDRAM_COLS + j)<<2);
      matrixB[i][j] = *(uint32_t*)address;
    }
  }
}
```

Listing 1.6: Read matrix from SDRAM in row-major order.

```
void SDRAM_mat_col_access_test(void){
  volatile uint32_t address;

    for (int i = 0; i<SDRAM_COLS; i++) {
    for(int j=0; j<MAT_ROWS; j++) {
      address = PA3_SDRAM_DEVICE_ADDR + ((j*SDRAM_COLS + i)<<2);
      matrixB[j][i] = *(uint32_t*)address;
    }
  }
}
```

Listing 1.7: Read matrix from SDRAM in column-major order.

Figure 1.40 illustrates one read issued from the CPU for the first scenario (row-major order access). The FMC SDRAM controller does not support SDRAM burst reads or writes (the only allowable burst length is 1). Instead, it supports burst reads on the CPUs AHB bus by utilizing the internal FIFO. Hence, it anticipates four READ commands to fill in the internal FIFO. The FIFO content is then transferred to the CPU using the AHB burst read of length 4.

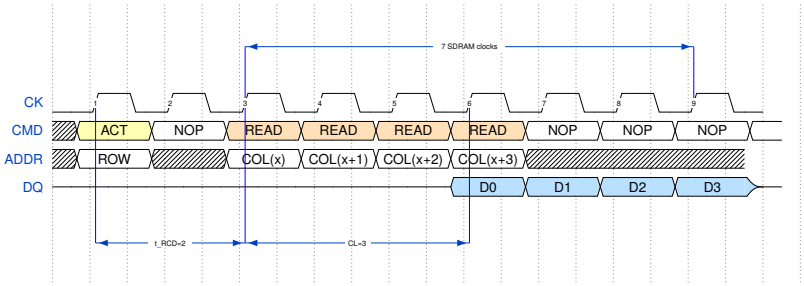

Fig. 1.40: Using row-major order to read a matrix, the SDRAM controller anticipates four consecutive READ command to the active SDRAM row for each read initiated from the CPU

In the second scenario, the matrix is accessed using column-major order. Figure 1.41 illustrates two consecutive reads issued from the CPU. As the CPU reads data from consecutive rows in each iteration, the CPU controller first reads four consecutive words from the active SRAM row and fills the internal FIFO, but it only returns one word to the CPU over the AHB bus. As the CPU starts another read from the next row, the SDRAM controller first precharges the active row. It then waits for two SDRAM clock periods (Row Precharge time) before activating the next row.

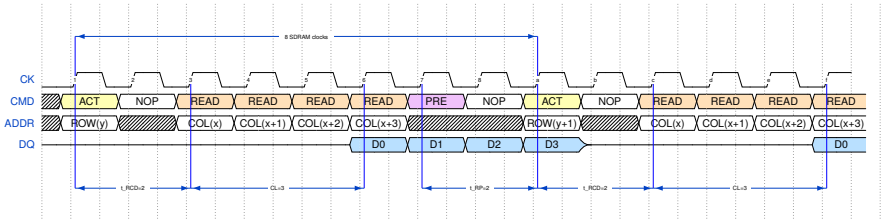

Fig. 1.41: Using column major order results in activating, reading and precharging an SDRAM row for every read issued from the CPU.

It is obvious that row-major order access is considerably faster than column-major order access. A rough estimate of the access time for row-major order access

considering an already open row is seven (7) SDRAM clock periods per four words. On the other side, a rough estimate of the access time for column-major order access is eight (8) SDRAM clock periods per word. Recall that only one word is transferred to the CPU, although the SDRAM controller anticipates four consecutive reads from the active row.

To assess the performance (speed) of the row-major and column-major matrix reads, we use the code in Listing 1.8. For each test, the code first sets the PC8 pin and reads the timer TIM3 counter value (this is the start of the test). After the test, we reset the PC8 pin and read the timer TIM3 counter value (this is the start of the test). By setting and resetting the PC8 pin, we can measure the duration of each test using an oscilloscope. The timer TIM3 runs at 1MHz (1 us resolution). Hence, we can estimate the duration of each test simply by reading the timer counter before and after the test.

```
// Row-major order access:
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_SET);
timer_val_start = __HAL_TIM_GET_COUNTER(&TIM3Handle);
SDRAM_mat_row_access_test();
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_RESET);
timer_val_end = __HAL_TIM_GET_COUNTER(&TIM3Handle);
if (timer_val_end > timer_val_start)
  elapsed_rows = timer_val_end - timer_val_start;
else
  elapsed_rows = timer_val_end + (65536-timer_val_start);

// Column-major order access:
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_SET);
timer_val_start = __HAL_TIM_GET_COUNTER(&TIM3Handle);
SDRAM_mat_col_access_test();
timer_val_end = __HAL_TIM_GET_COUNTER(&TIM3Handle);
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_RESET);
if (timer_val_end > timer_val_start)
  elapsed_cols = timer_val_end - timer_val_start;
else
  elapsed_cols = timer_val_end + (65536-timer_val_start);
```

Listing 1.8: Code used to test the speed of row-major and column-major matrix read from the SDRAM.

Figure 1.42 shows the oscilloscope trace for the signal on the GPIOC pin. It shows that the row-major order read lasts for about 2.3 ms, while the column-major order read lasts for about 10 ms. Using the timer counter, we estimate the duration of the row-major order read to 2365 us and the duration of the column-major order read to 9816 us. Both measurements show that the row-major order read is about four times faster than the column-major order read, which is in accordance with the rough estimation from figures 1.40 and 1.41.

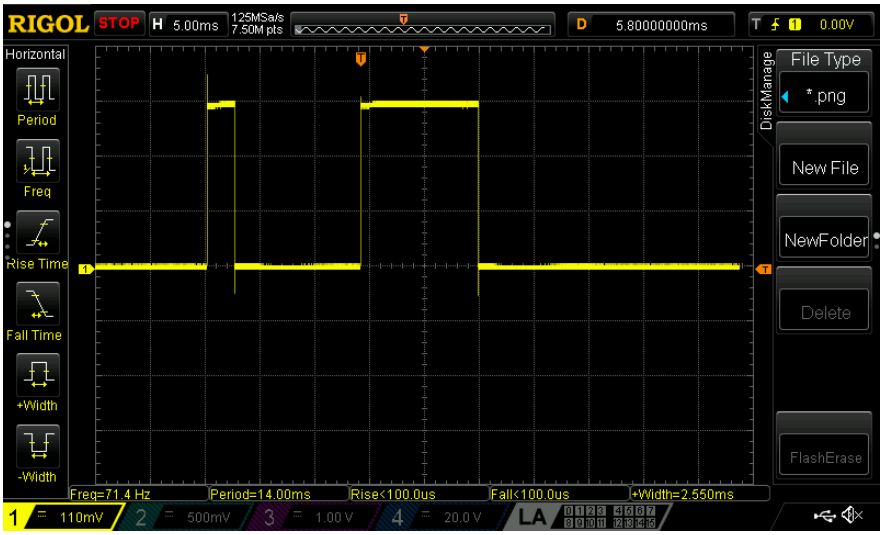

Fig. 1.42: Oscilloscope trace on the GPIOC pin 8. The row-major order matrix read lasts for about 2.5 ms while the column-major order matrix read lasts for more than 10 ms.

1.9.3.1 Using DMA to transfer data from an external SDRAM to the internal SRAM

Direct memory access (DMA) is used to provide high-speed data transfer between peripherals and memory and between memory and memory without any CPU action (except DMA controller initialization and DMA transfer request in case of memory-to-memory transfer). As already described in Section 4.5.2, the DMA controller in STM32 microcontrollers (actually, there are two DMA controllers, DMA1 and DMA2, respectively) have 16 streams in total (8 for each DMA controller), each dedicated to managing memory access requests from one or more peripherals. Each stream can have up to 8 channels (requests) in total. Each DMA controller has an arbiter for handling the priority between DMA requests. According to the STM32F69I reference manual, the memory-to-memory mode in DMA is a mode that doesn't need any triggering request from a peripheral, and it will happen just after the stream enable bit is set. Also, according to the STM32F69I reference manual, only the DMA2 could handle memory-to-memory data transfers. The stream can be enabled just by setting the Enable bit (EN) in the DMA SxCR register. Then, the stream immediately fills the FIFO up to the threshold level. When the threshold level is reached, the FIFO contents are drained and stored in the destination.

Before using the DMA2 controller to transfer data from one memory region to another, we must configure (initialize) the DMA2 controller as described in Section 4.5.2. When configuring the DMA controller we:

1. Select a stream that we wish to use. Any available stream in the DMA2 controller can be used for memory-to-memory transfers.
2. Select a channel; this is irrelevant for memory-to-memory transfers because a peripheral device does not trigger the DMA transfer through a channel. Instead, it is triggered by setting the EN bit in the DMA SxCR register.
3. Set a priority for a selected DMA stream.
4. Set the number of data to be transferred (it can be any value from 1 to 65535).
5. Set the source and destination transfer width (byte, half-word, word).
6. Set the source and destination addresses.
7. Select whether the source and destination addresses should be incremented during the transfer. For memory-to-memory transfers, both addresses should be incremented during the transfer.
8. Select whether the burst transfers of 4, 8 or 16 beats should be used during the transfer.

Programming DMA is relatively easy. Recall from Section 4.5.2 that each stream can be controlled using four registers: memory address register, peripheral address register, number of data register, and configuration register. Once set, DMA takes care of memory address increment without disturbing the CPU. Now that it is clear how the DMA works from a theoretical point of view, we can use the HAL library to configure and use a DMA controller. The HAL library abstracts most of the underlying hardware details. The DMA controller is abstracted in HAL with a C structure `DMA_HandleTypeDef`. Let us describe more in-depth only the two most important fields of this structure:

- `Instance`: this is the pointer to the DMA Stream descriptor we will use. For example, `DMA2_Stream1` indicates the first stream of DMA2. The stream descriptor is a C structure that contains all DMA stream registers. The reference to the Instance structure points to the actual peripheral address. For example, the `DMA2_Stream1` is defined in HAL as a pointer to the stream descriptor structure, and it holds the register base address for DMA2 Stream1 registers.
- `Init`: is an instance of the C structure `DMA_InitTypeDef`, which is used to configure the DMA Stream and channel.

`DMA_InitTypeDef` is defined in the following way:

```
typedef struct
{
  uint32_t Channel;
  uint32_t Direction;
  uint32_t PeriphInc;
  uint32_t MemInc;
  uint32_t PeriphDataAlignment;
  uint32_t MemDataAlignment;
  uint32_t Mode;
  uint32_t Priority;
  uint32_t FIFOMode;
  uint32_t FIFOThreshold;
  uint32_t MemBurst;
  uint32_t PeriphBurst;
}DMA_InitTypeDef;
```

Listing 1.9: DMA `DMA_InitTypeDef` C structure.

Let us briefly describe the C `DMA_InitTypeDef` structure:

- `Channel`: Specifies the channel used for the specified stream. It can assume the values DMA_CHANNEL_0, DMA_CHANNEL_1 up to DMA_CHANNEL_7. The peripherals are bound to streams and channels during the MCU design, so we should consult the datasheet for our microcontroller to see the stream/channel bound to the peripheral we want to use with DMA.
- `Direction`: Specifies if the data will be transferred from memory-to-peripheral, memory-to-memory or peripheral-to-memory.
- `PeriphInc`: Specifies whether the Peripheral address register should be incremented or not during the DMA transfer. Recall that a DMA controller has one peripheral port used to specify the address of the peripheral register involved in the DMA transfer. Since a DMA transfer usually involves several bytes, the DMA can be configured to increment the peripheral register for every transmitted byte.
- `MemInc`: Specifies whether the memory address register should be incremented or not during the DMA transfer.
- `PeriphDataAlignment`: Specifies the Peripheral data width. Transfer data sizes of the peripheral and memory are fully programmable through this field and the next one. The DMA controller is designed to automatically perform data alignment when source and destination data sizes differ.
- `MemDataAlignment`: Specifies the Memory data width.
- `Mode`: the DMA controller has two working modes: normal and circular. In normal mode, the DMA sends the specified amount of data from the source to the destination port and stops the activities. It must be re-activated again to do another transfer. In circular mode, it automatically resets the transfer counter at the end of transmission and starts transmitting again from the first byte of the source buffer.
- `Priority`: Specifies the software priority for the DMA Stream. The priority allows the internal arbiter in the DMA controller to rule concurrent requests.
- `FIFOMode`: Specifies if the stream uses the FIFO buffer. Recall that each stream has an independent 4-word (4 * 32 bits) FIFO. The FIFO temporarily stores data coming from the source before transmitting it to the destination. The FIFO introduces one important advantage: it reduces S(D)RAM access time by supporting burst transactions. The FMC SDRAM controller used in our case issues four READ commands in a row, thus reading four consecutive words from an active SDRAM row. Using the FIFO allows for storing these four words efficiently before they are sent to SRAM.
- `FIFOThreshold`: Specifies the FIFO threshold level. The FIFO will be drained to the destination when this threshold is achieved.
- `MemBurst`: Specifies the amount of data to be transferred to/from memory in a single non-interruptible transaction.

- `PeriphBurst`: Specifies the amount of data to be transferred to/from peripheral (or memory for mem-to-mem DMA transfers) in a single non-interruptible transaction.

All HAL functions related to DMA manipulation are designed so that they accept as the first parameter an instance of the C structure `DMA_HandleTypeDef`. To initialise the DMA Stream, we first set all desired parameters in the `DMA_InitTypeDef` structure and then use the HAL function `HAL_DMA_Init(DMA_HandleTypeDef *hdma)`. The following code illustrates configuring and initialising the DMA2 Stream 1 for memory-to-memory transfers using FIFO and burst of length 4:

```
HAL_StatusTypeDef DMA2_SDRAM_Config(DMA_HandleTypeDef* DmaHandle)
{
  /* Enable DMA2 clock */
  __HAL_RCC_DMA2_CLK_ENABLE();

  /* Select the DMA Stream to be used #*/
  DmaHandle->Instance = DMA2_Stream1;

  /* Set the DMA Parameters */
  /* DMA_CHANNEL_0                    */
  DmaHandle->Init.Channel = DMA_CHANNEL_0;
  /* M2M transfer mode                */
  DmaHandle->Init.Direction = DMA_MEMORY_TO_MEMORY;
  /* Peripheral increment mode Enable */
  DmaHandle->Init.PeriphInc = DMA_PINC_ENABLE;
  /* Memory increment mode Enable     */
  DmaHandle->Init.MemInc = DMA_MINC_ENABLE;
  /* Peripheral data alignment : Word */
  DmaHandle->Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
  /* memory data alignment : Word     */
  DmaHandle->Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
  /* Normal DMA mode                  */
  DmaHandle->Init.Mode = DMA_NORMAL;
  /* priority level : high            */
  DmaHandle->Init.Priority = DMA_PRIORITY_HIGH;
  /* FIFO mode enabled                */
  DmaHandle->Init.FIFOMode = DMA_FIFOMODE_ENABLE;
  /* FIFO threshold: full             */
  DmaHandle->Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
  /* Memory burst                     */
  DmaHandle->Init.MemBurst = DMA_MBURST_INC4;
  /* Peripheral burst                 */
  DmaHandle->Init.PeriphBurst = DMA_PBURST_INC4;

  /* Initialize the DMA stream */
  if (HAL_DMA_Init(DmaHandle) != HAL_OK)
  {
    /* Initialization Error */
    return HAL_ERROR;
  }

  /* Configure NVIC for DMA transfer complete/error interrupts */
  HAL_NVIC_SetPriority(DMA2_Stream1_IRQn, 0, 0);

  /* Enable the DMA STREAM global Interrupt */
  HAL_NVIC_EnableIRQ(DMA2_Stream1_IRQn);

  return HAL_OK;
}
```

Listing 1.10: DMA2 Controller configuration and initialization.

In the above code, the DMA2 Stream 1 is configured to automatically increment the source and destination addresses for each transmitted word. Remember that we are transferring the block of contiguous memory words in the matrix from the external SDRAM to the matrix in the internal SRAM. Hence, the addresses in the source and destination block should increase for each word transferred. Also, we enable the FIFO, set the FIFO threshold level to full and enable the burst transfers of size 4. In such a way, the DMA controller will read four words from SDRAM to the FIFO and then transfer them to SRAM.

After DMA initialisation, we should set the priority for the DMA2 Stream 1 interrupt and enable the interrupt request generated by DMA2 Stream 1. The DMA2 Controller will then assert an interrupt request whenever the DMA 2 Stream 1 completes the DMA transfer. Hence, we should also implement the minimal interrupt handler for the DMA2 Stream 1 as follows:

```
void DMA2_Stream1_IRQHandler(void)
{
  /* Check the interrupt and clear flag */
  HAL_DMA_IRQHandler(&DMA2_SDRAM_Handle);
}
```

Listing 1.11: DMA2 Stream 1 Interrupt Handler.

Now we should create a DMA handle variable and initialize the DMA2 Stream 1 by passing reference to the DMA handle into the `DMA2_SDRAM_Config` function:

```

DMA_HandleTypeDef     DMA2_SDRAM_Handle;

...

// Configure DMA2 for SDRAM:
if (DMA2_SDRAM_Config(&DMA2_SDRAM_Handle) != HAL_OK) {
   Error_Handler();
}
```

Listing 1.12: DMA2 Stream 1 handle and its configuration.

To initiate the memory-to-memory DMA transfer, we use `HAL_DMA_Start(DMA_HandleTypeDef *hdma, uint32_t SrcAddress, uint32_t DstAddress, uint32_t DataLength)`. The arguments are the pointer to the DMA handle, the source address, the destination address and the number of data to transfer. This function sets the EN bit in DMA SxCR, which in turn triggers the DMA controller to start the transfer. Hence, to transfer a matrix from the external SDRAM to SRAM, we implement and use the following function:

```
void SDRAM_DMA_mat_row_access_test(void){
  volatile uint32_t address;

  for (int k = 0; k < N; k++)
  {
    HAL_DMA_Start(&DMA2_SDRAM_Handle,
                  (uint32_t) PA3_SDRAM_DEVICE_ADDR_RW,
```

```
                        (uint32_t) matrixB,
                        MAT_ROWS * SDRAM_COLS);
    HAL_DMA_PollForTransfer(&DMA2_SDRAM_Handle,
                                HAL_DMA_FULL_TRANSFER,
                                HAL_MAX_DELAY);
  }
}
```

Listing 1.13: Matrix transfer using DMA.

The function `HAL_DMA_PollForTransfer()` waits for DMA transfer to complete. Otherwise, the CPU would continue to execute the program and would not bother with DMA transfer (which would be the desired way), but in our case, we are going to measure the time required to transfer the matrix from SDRAM to SRAM using DMA; hence, we should wait for DMA to terminate. The function `HAL_DMA_PollForTransfer()` is used here for the sake of simplicity, but it is strongly recommended to use the DMA interrupt handler instead. Finally, we can add the DMA matrix transfer to a set of the previous performance tests in Listing 1.8 as follows:

```
// Row-major order access:
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_SET);
timer_val_start = __HAL_TIM_GET_COUNTER(&TIM3Handle);
SDRAM_mat_row_access_test();
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_RESET);
timer_val_end = __HAL_TIM_GET_COUNTER(&TIM3Handle);
if (timer_val_end > timer_val_start)
  elapsed_rows = timer_val_end - timer_val_start;
else
  elapsed_rows = timer_val_end + (65536-timer_val_start);

// Column-major order access:
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_SET);
timer_val_start = __HAL_TIM_GET_COUNTER(&TIM3Handle);
SDRAM_mat_col_access_test();
timer_val_end = __HAL_TIM_GET_COUNTER(&TIM3Handle);
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_RESET);
if (timer_val_end > timer_val_start)
  elapsed_cols = timer_val_end - timer_val_start;
else
  elapsed_cols = timer_val_end + (65536-timer_val_start);

// DMA transfer:
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_SET);
timer_val_start = __HAL_TIM_GET_COUNTER(&TIM3Handle);
SDRAM_DMA_mat_row_access_test();
timer_val_end = __HAL_TIM_GET_COUNTER(&TIM3Handle);
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8, GPIO_PIN_RESET);
if (timer_val_end > timer_val_start)
  elapsed_cols = timer_val_end - timer_val_start;
else
  elapsed_cols = timer_val_end + (65536-timer_val_start);
```

Listing 1.14: Code used to test the speed of row-major, column-major and DMA matrix read from the SDRAM.

When executing the DMA performance test, we observe from Figure 1.43 that the time required to transfer the matrix from the external SDRAM to the internal SRAM is only about 1500 us. Why is the DMA controller faster than the CPU, considering

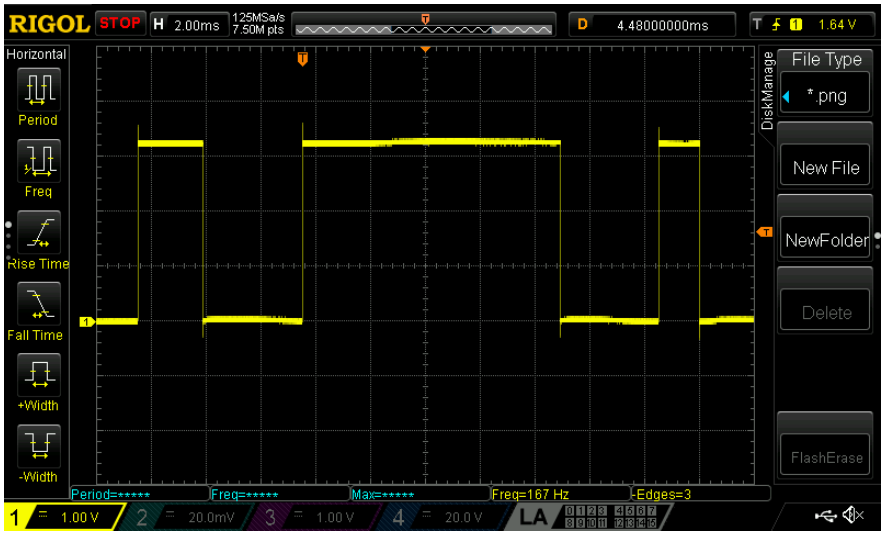

Fig. 1.43: Oscilloscope trace on the PC8 pin. The row-major order matrix read lasts for about 2.5 ms, the column-major order matrix read lasts forabout 10 ms while DMA transfer lasts for about 1.5 ms.

that the same amount of data is being transferred from/to the same devices in both cases?

```
J_LOOP:
  ; address = PA3_SDRAM_DEVICE_ADDR_RW  + ((i*SDRAM_COLS + j)<<2);
  add.w    r1, r3, #0           ; r1 <- r3
  ldr      r2, [pc, #60]        ; r2 <- 0xC0008000 (SDRAM address)
  add.w    r2, r2, r1, lsl #2   ; r2 <- r2+(r1*4) LOAD FROM SDRAM
  ; matrixB[i][j] = *(uint32_t*)address;
  ldr      r0, [r2, #0]         ; r0 <- M_SDRAM[r2]
  ldr      r2, [pc, #52]        ; r2 <- matB base address
  str.w    r0, [r2, r1, lsl #2] ; matB[i][j] <- r0  STORE TO SRAM
  ; for(int j=0; j<SDRAM_COLS; j++) {
  adds     r3, #1               ; inc r3 (r3 holds j)
  cmp      r3, #255             ; if j <= 255
  ble.n    J_LOOP               ;       loop back
```

Listing 1.15: Assembly code corresponding to the instructions created by the compiler for the innermost loop in Listing 1.6. There are 11 instructions executed in each iteration of the innermost loop; hence 11 instructions are executed for transferring one word from SDRAM to SRAM. The first four instructions are used to calculate the address in SDRAM. Then, four instructions are used to read the word from SDRAM and write it to SRAM, and finally, the last three instructions increments the innermost loop counter, compare it to 255 and loop if not equal.

Well, the answer lies in the fact that the DMA controller does not execute instructions. For each word transferred, the CPU fetches the `LDR` instruction (load register

with word), executes it (it loads the data from SDRAM to an internal register), fetches the `STR` instruction (store register as word), and finally executes it (it stores the data from the internal register to SRAM). Besides LDR and STR instructions, in each loop iteration, the CPU executes a bunch of other instructions required to calculate the address in SDRAM, increment and compare the loop index, etc. (see Listing 1.15). The DMA controller only transfers data from SDRAM (in bursts!) and forwards them to SRAM (in bursts!) without fetching and executing the load-/store instructions! Besides offloading the CPU, this is another benefit of utilizing DMA controllers.