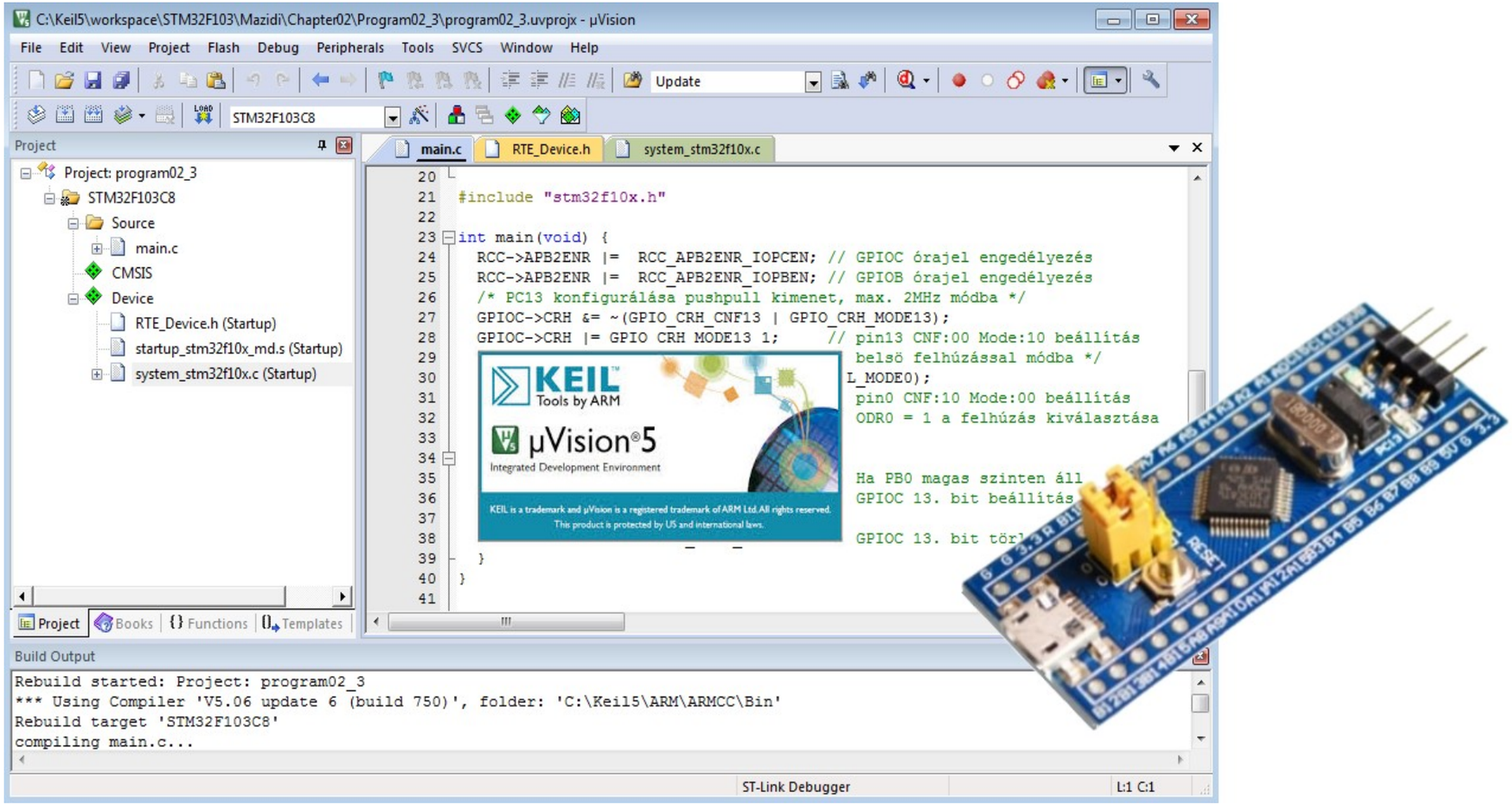
STM32 mikrovezérlők programozása ARM Keil környezetben



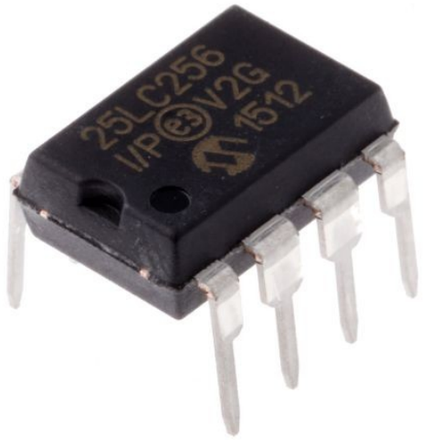
14. Soros Periféria Illesztő (SPI) – 3. rész

Felhasznált és ajánlott irodalom

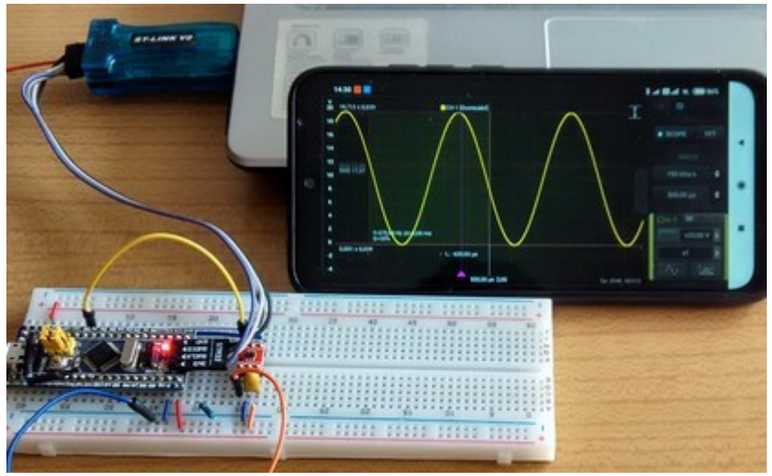
- Joseph Yiu: Cortex-M for Beginners
- Joseph Yiu: The Definitive Guide To The ARM CORTEX-M3
- Muhammad Ali Mazidi, Shujen Chen, Eshragh Ghaemi: STM32 Arm Programming for Embedded Systems
- Alexander Tarasov: **Курс «Штурмуем STM32»**
- Warren Gay: Beginning STM32 - Developing with FreeRTOS, libopencm3 and GCC
- ARM Keil MDK Getting started
- **STM32F103C8** adatlap és termékinfo
- **STM32F103** Family Reference Manual

Témák, mintaprogramok

- **Program08_3** – kétirányú SPI adatátvitelt mutatunk be egy Microchip gyártmányú, **25LC256** típusú EEPROM memória írása és olvasása kapcsán. A program futása során 64 bájtos memórialapokat írunk, illetve olvasunk vissza. A visszaolvasott szöveget az **SWO** kimeneten keresztül kiíratjuk (csak debug módban látjuk)

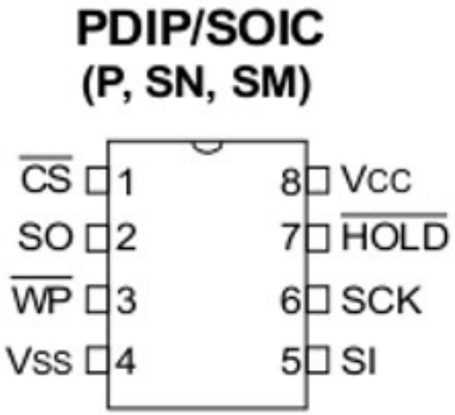


- **Program08_5** – Adatküldés **DMA**-val az **SPI1** csatornán keresztül, az előző előadásban bemutatott **DAC7512** 12-bites digitális-analóg átalakítónak. A program egy szinuszhullámot tartalmazó táblázatból küld ki adatokat egyenletes időközönként, **Timer3**-mal ütemezve. Természetesen a táblázat tetszés szerinti hullámforma adatait tartalmazhatná



A 25LC256 SPI EEPROM használata

- A **Microchip 25LC256** típusú EEPROM memória 256 Kbit kapacitású (32Kx8 bit)
- Tápfeszültség: 2,5 - 5,5 V, az SPI órajel max. 10 MHz lehet. A sebesség azonban tápfeszültség függő: az adatlap szerint 2,5 - 4,5 V között csak max. 5 MHz lehet
- Újraírhatósági szám: 1 millió
- Minden tranzakció egy parancsbájt kiküldésével kezdődik
  Az elfogadott parancsokat az alábbi táblázatban foglaltuk össze:

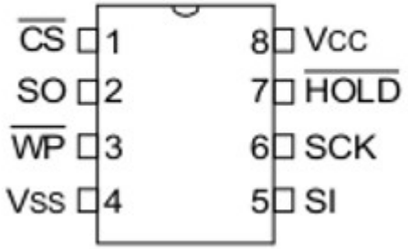


Parancs	Kód	A funkció rövid leírása
READ	**0x03**	A memória olvasása adott címtől kezdődően
WRITE	**0x02**	A memória írása adott címtől kezdődően
WRDI	**0x04**	Az írást engedélyező bit törlése (letiltja az írást)
WREN	**0x06**	Az írást engedélyező bit beállítása (engedélyezi az írást)
RDSR	**0x05**	A **STATUS** regiszter olvasása
WRSR	**0x01**	A **STATUS** regiszter írása

A 25LC256 EEPROM STATUS regisztere

- A **STATUS** regiszter állapotjelző biteket tartalmaz amelyek az IC aktuális állapotáról értesítenek bennünket a működés során. A legalsó két bit csak olvasható, a többi bit írható/olvasható. Az egyes bitek jelentését az alábbiakban foglalhatjuk össze:

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
WPEN	-	-	-	BP1	BP0	WEL	WIP



- **WPEN** – a hardveres írásvédelem engedélyezése (**1:** WP engedélyezve, **0:** WP letiltva)
- **BP1, BP0** – blokk védelem beállítása (**00:** nincs tiltás, **01:** a memória felső negyede (0x6000 címtől) írásvédett, **10:** a felső fele (0x4000 címtől) írásvédett, **11:** a teljes memória írásvédett)
- **WEL** – Ez a bit jelzi az írás engedélyezés (Write Enable Latch) állapotát. Csak olvasható, illetve a **WRDI/WREN** utasításokkal befolyásolható (**1:** az írás engedélyezett, **0:** írás letiltva)
- **WIP** - Írás folyamatban (Write In Progress) jelzőbit, csak olvasható (**1:** írás folyamatban, **0:** írás befejeződött)

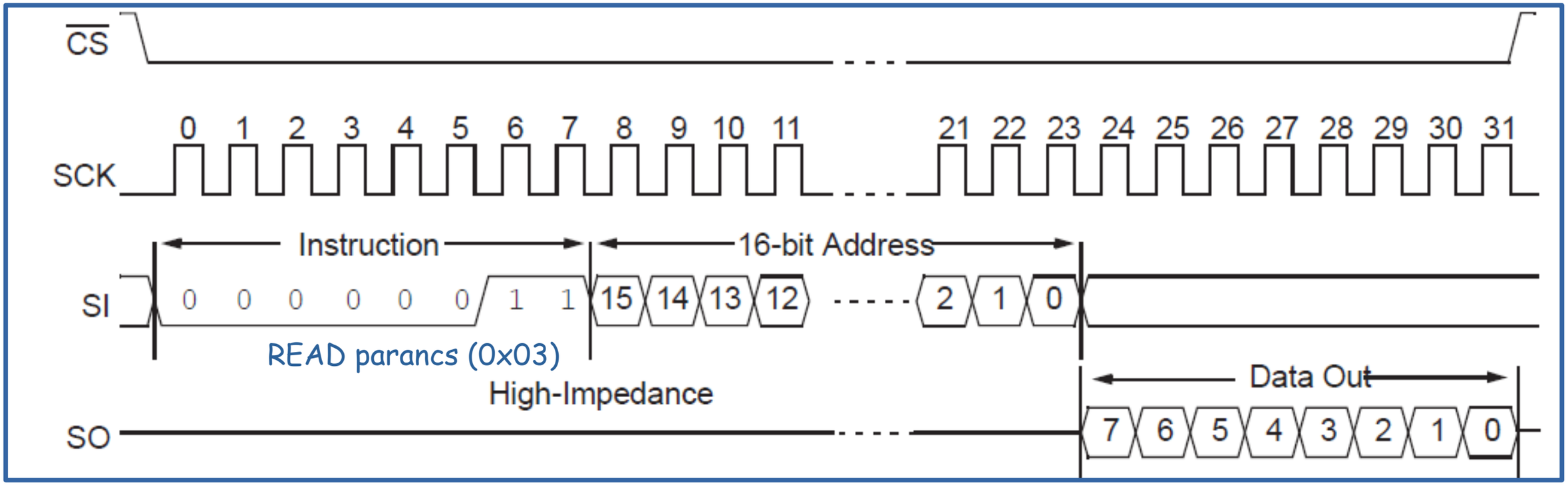
A 25LC256 EEPROM írásvédelmi rendszere

- Az írást engedélyező retesz (**WEL**) törlődik bekapcsoláskor
- Egy írást engedélyező parancsot (**WREN**) kell kiadni minden írási művelet előtt
- Minden bájt, memórialap vagy a **STATUS** regiszter írása után az írást engedélyező retesz (**WEL**) automatikusan törlődik (a következő íráshoz újra kell engedélyezni)
- Az adatküldés végén a **CS** jelet magas szintre kell húzni, ez indítja el a belső írás műveletet
- Az írásvédelmi rendszer működését az alábbi táblázatban foglaltuk össze. Látható, hogy a **STATUS** regiszter írásával beállítható **WPEN** bit a hardveres írásvédelem (a **WP** bemenet) engedélyezésére szolgál.

WEL	WPEN	WP	Védett blokkok	Nem védett blokkok	Státus regiszter
0	x	x	írásvédett	írásvédett	írásvédett
1	0	x	írásvédett	írható	írható
1	1	L	írásvédett	írható	írásvédett
1	1	H	írásvédett	írható	írható

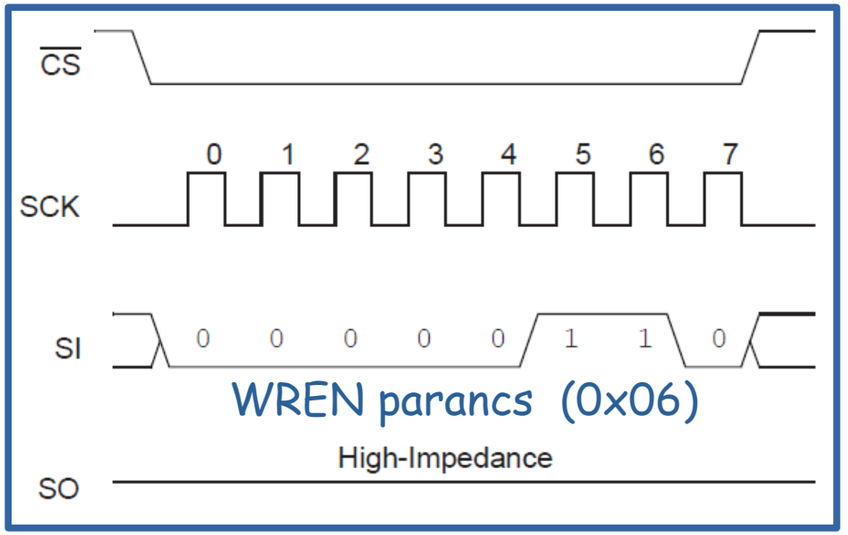
Adat olvasási szekvencia

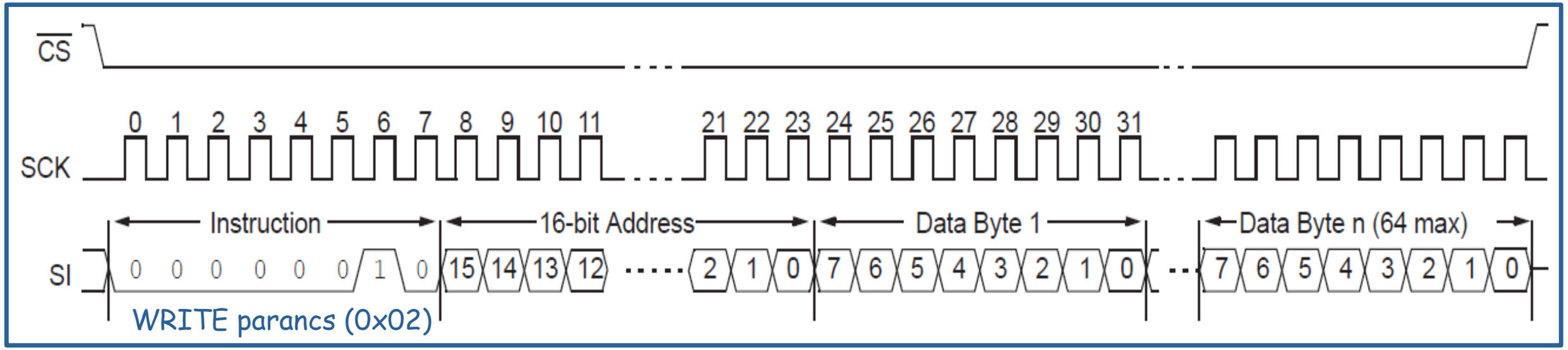
- A parancsbájt és a kétbájtos cím kiküldését követően egy vagy több bájtot olvashatunk, lehetőség van ákár 64 bájt (egy teljes memórialap) olvasására is.
- A 16 bites cím legelső (legnagyobb helyiértékű) bitjének tartalma közömbös, mivel a 32 K címzése csak 15 bitet használ



Adat írási szekvencia

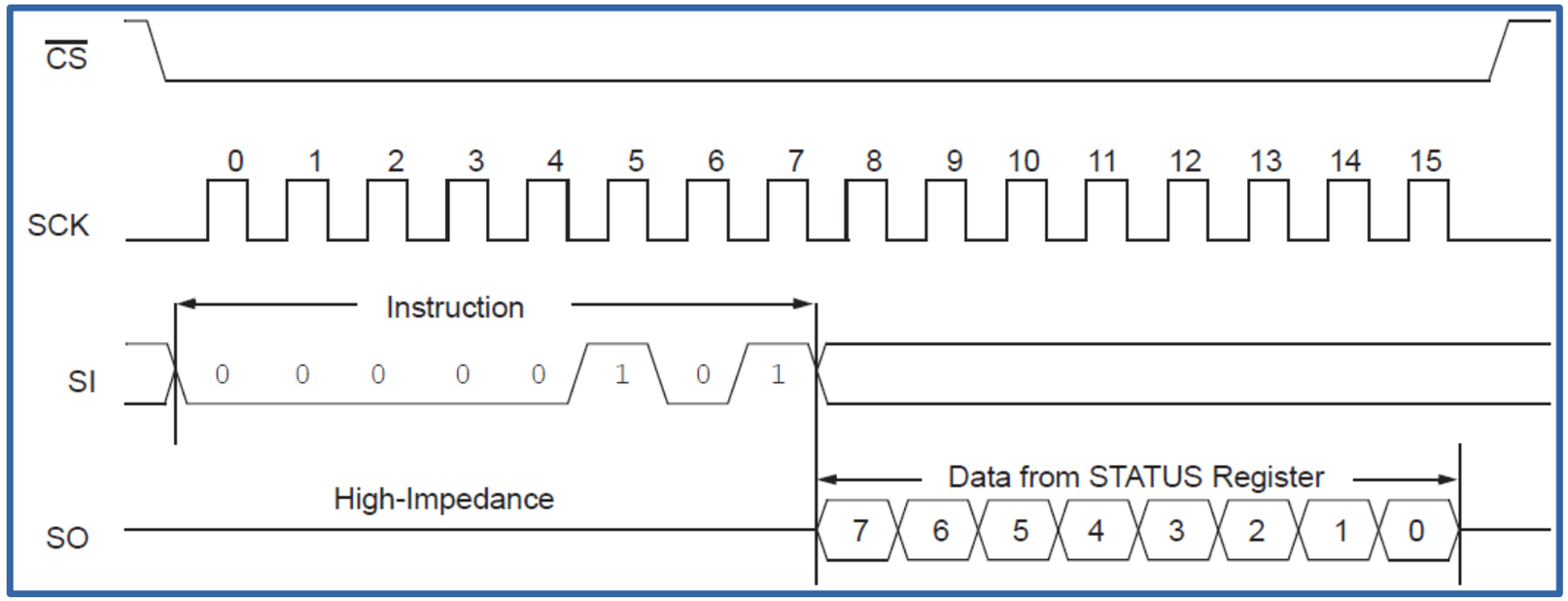
- Az írás megkezdése előtt egy **WREN** parancsot kell kiküldenünk az írás (újra)engedélyezéséhez
- Az írási szekvencia hasonló az olvasáshoz: a **WRITE** parancsbájtot követően ki kell küldenünk a 16 bites kezdőcímet (a legfelső bit nem használt), majd egymás után kiküldjük az adatbájtokat (a cím automatikusan inkrementálódik)
- Max. 64 bájtot írhatunk, és laphatárt nem léphetünk át





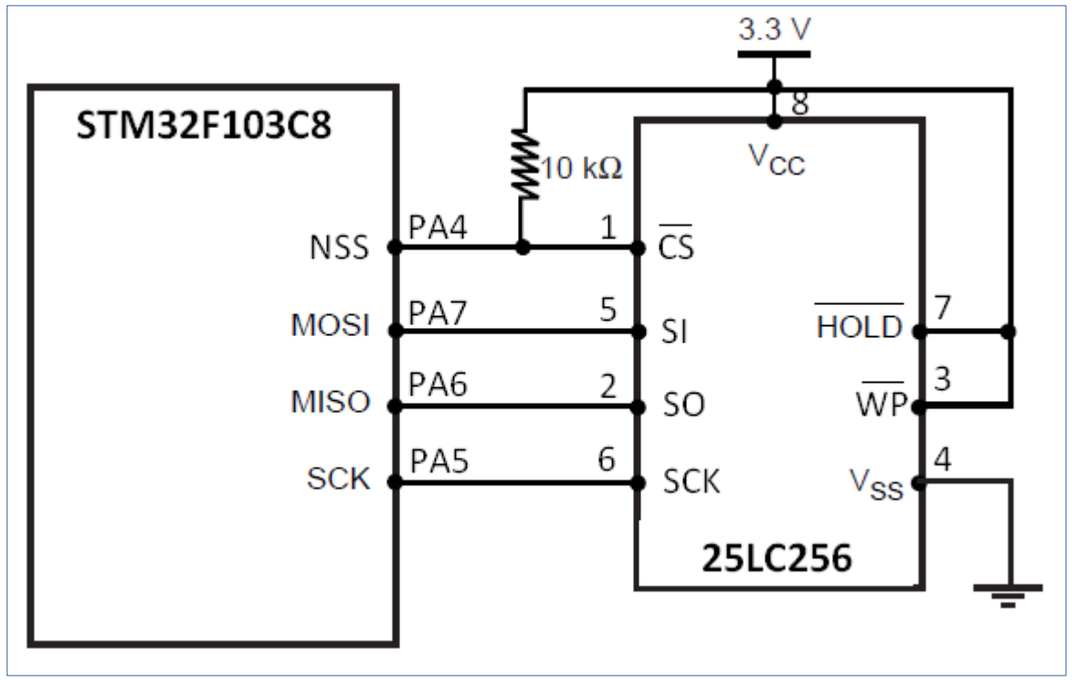
A foglaltság ellenőrzése

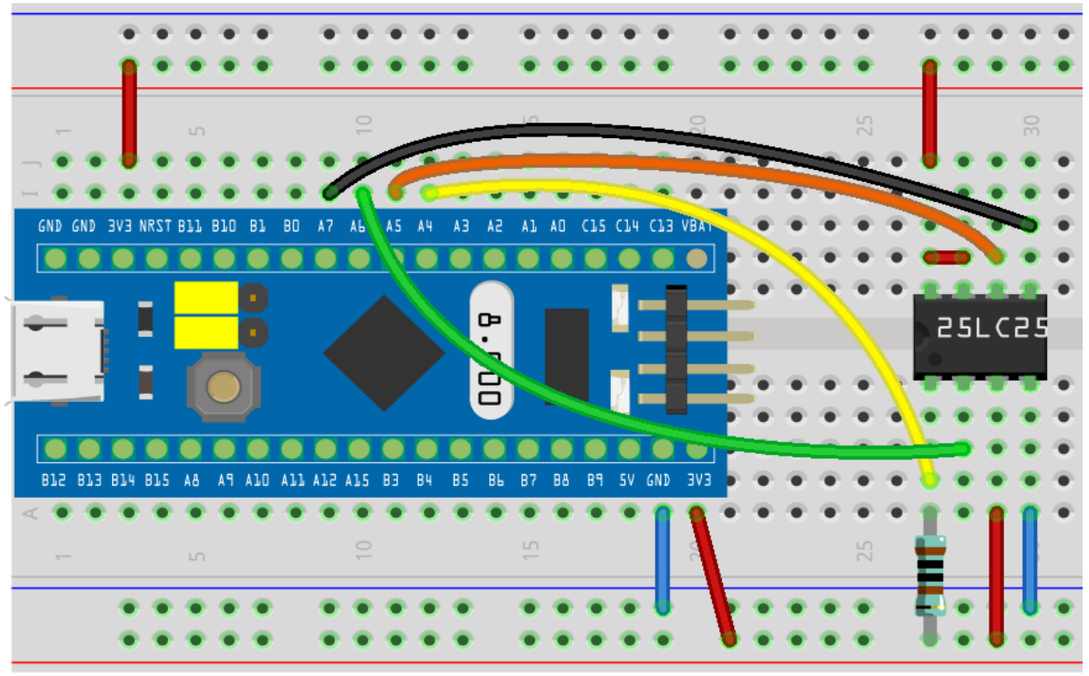
- Az írás művelet belső időzítésű és hosszabb időt (akár 5 ms) vesz igénybe
  Ezalatt más parancsot nem fogad az IC, csak a **STATUS** regiszter olvasását
- Fentiek miatt minden művelet megkezdése előtt vizsgálni kell a foglaltságot:
  - Az **RDSR** paranccsak kiolvassuk a **STATUS** regiszter tartalmát
  - Megvizsgáljuk a **WIP** (Write In Progress – írás folyamatban) bitet,
- A fenti lépéseket addig ismételjük, amíg a **WIP** bit értéke 0 nem lesz



A kísérleti kapcsolás vázlata

- Az **SPI1** csatornát használjuk (**PA4:** NSS, **PA5:** SCK, **PA6:** MISO, **PA7:** MOSI)
- A **WP** (írásvédelem) és a **HOLD** (művelet felfüggesztése) bemeneteket most nem használjuk, ezeket magas szintre kell felhúzni
- Az **NSS** kimenet (**PA4**) csak lefelé húz (OD kimenet), ezért kell egy felhúzó ellenállás a **CS** bemenetre, értéke azonban befolyásolja a sebességet *(pl. 1 kΩ @ 2.25 MHz bevált)*





Program08_3/main.c 6/1. oldal

```
#include "stm32f10x.h"
//----- print átirányítás ------------
#include <stdio.h>
#pragma import(__use_no_semihosting_swi)
struct __FILE { int handle; };
FILE __stdout;
FILE __stdin;

int fputc(int ch, FILE *f) {
  ITM_SendChar(ch);
  return (ch);
}

void _sys_exit(int return_code) {
  label:  goto label;  /* endless loop */
}
//--- print átirányítás vége ---------
```

printf átirányítás SWO kimenetre

- Egy **25LC256** típusú EEPROM írása és olvasása **SPI1** csatornán keresztül
- A program futása során 64 bájtos memórialapokat írunk, illetve olvasunk vissza
- A visszaolvasott szöveget az **SWO** kimeneten kiíratjuk (csak debug módban látjuk)

```
#define BLKSIZE      64                // blokkméret: egy memórialap 64 bájtos
//-- A 25LC256 EEPROM által elfogadott parancsok --------
#define CMD_READ  0x03                 // Olvasás a megadott címtől kezdődően
#define CMD_WRITE 0x02                 // Írás a megadott címtől kezdődően
#define CMD_WRDI  0x04                 // Letiltja az írást
#define CMD_WREN  0x06                 // Engedélyezi az írást
#define CMD_RDSR  0x05                 // Státuszregiszter olvasása
#define CMD_WRSR  0x01                 // Státuszregiszter írása
```

Folytatás a következő oldalon ...

Program08_3/main.c 6/2. oldal

```
//-- Slave Select kimenet kezelése -------
#define SPI1_ENABLE()        SPI1->CR1 |=  SPI_CR1_SPE;         // SPI1 engedélyezés
#define SPI1_DISABLE()       SPI1->CR1 &= ~SPI_CR1_SPE;         // SPI1 letiltás

/*-------------------------------------------------------------------------
    Az SPI1 csatorna konfigurálása 8 bites módba, 2.25 MHz bitrátával
    Az alapértelmezett PA4=SS, PA5=SCK, PA7=MOSI kivezetéseket használjuk
        Az NSS kimenet vezérlése hardveresen történik
 *-------------------------------------------------------------------------*/
void SPI1_init(void) {
  RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;  // SPI1 órajel engedélyezés
  RCC->APB2ENR |= RCC_APB2ENR_AFIOEN;  // AFIO órajel engedélyezés
  RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;  // GPIOA órajel engedélyezés
  AFIO->MAPR &= ~AFIO_MAPR_SPI1_REMAP; // Törli SPI1 REMAP bitjét

  /* PA5, PA6, PA7 konfigurálása, mint SPI1 SCK, MISO és MOSI */
  /* PA4 konfigurálás, mint SPI1 NSS  */
  GPIOA->CRL &= ~0xFFF00000;           // CNF és MODE bitek törlése
  GPIOA->CRL |=  0xB8BF0000;           // CNF=10 MODE=11 (PA5,PA7) AltFunc puspull
  GPIOA->ODR |=  GPIO_ODR_ODR6;        // CNF=10 MODE=00 (PA6) input with pullup
                                       // CNF=11 MODE=11 (PA4)   AltFunc OD
  /* SPI1 konfigurálás: 8-bit, MSB first, HW NSS, Master mód */
  SPI1->CR1 = SPI_CR1_BR_2 |           // Baud rate 72/32 = 2.25 MHz
              SPI_CR1_MSTR;            // Master mód beállítás
  SPI1->CR2 = SPI_CR2_SSOE;            // Single Master mód
}
```

Folytatás a következő oldalon ...

Program08_3/main.c 6/3. oldal

```
/*-------------------------------------------------------------------------
    8 bit adat kiküldése és fogadása az SPI1 csatornán
 *-------------------------------------------------------------------------*/
uint8_t ioMasterSPI1(uint8_t data) {
  while (!(SPI1->SR & SPI_SR_TXE)) {}  // TXE jelre várunk
  SPI1->DR = data;                     // Adat küldése
  while (SPI1->SR & SPI_SR_BSY) {}     // Átvitel végére várunk
  return SPI1->DR;
}

/*-------------------------------------------------------------------------
   Várakozás arra, hogy az EEPROM befejezze az írást.
 *-------------------------------------------------------------------------*/
void waitFor25LC256(void) {
  uint8_t u8_flag;
  do {
    SPI1_ENABLE();                     // kiadjuk a Slave Select jelet
    ioMasterSPI1(CMD_RDSR);            // Státuszregiszter olvasása parancs
    u8_flag = ioMasterSPI1(0x00);      // Státuszregiszter olvasása adat
    SPI1_DISABLE();                    // megszüntetjük a Chip Enable jelet
  } while (u8_flag & 0x01);
}
```

Folytatás a következő oldalon ...

Program08_3/main.c 6/4. oldal

```
/** Egy memórialap (64 bájt) írása a bemenő adatbufferből, egy megadott címétől kezdődően.
 *  \param u16_MemAddr a memórialap kezdőcíme, ahová írunk
 *  \param *pu8_buf mutató az adatbuffer kezdetéhez
 */
void memWrite25LC256(uint16_t u16_MemAddr, char *pu8_buf) {
  uint8_t u8_i,u8_AddrLo, u8_AddrHi;
  u8_AddrLo = u16_MemAddr & 0x00FF;
  u8_AddrHi = (u16_MemAddr >> 8);
  WaitFor25LC256();                       // Várunk, ha az EEPROM elfoglalt
  SPI1_ENABLE();
  ioMasterSPI1(CMD_WREN);                 // Írás újraengedélyezése
  SPI1_DISABLE();
    waitFor25LC256();
  __nop(); __nop(); __nop();              // Kis várakozás
  SPI1_ENABLE();
  ioMasterSPI1(CMD_WRITE);                // Adatblokk írása
  ioMasterSPI1(u8_AddrHi);                // Kezdőcím megadása
  ioMasterSPI1(u8_AddrLo);
  for(u8_i=0; u8_i < BLKSIZE; u8_i++) {
    ioMasterSPI1(*pu8_buf++);             // Az adatbuffer kiírása
  }
  SPI1_DISABLE();                         // Itt indul az írási folyamat
}
```

Folytatás a következő oldalon ...

Program08_3/main.c 6/5. oldal

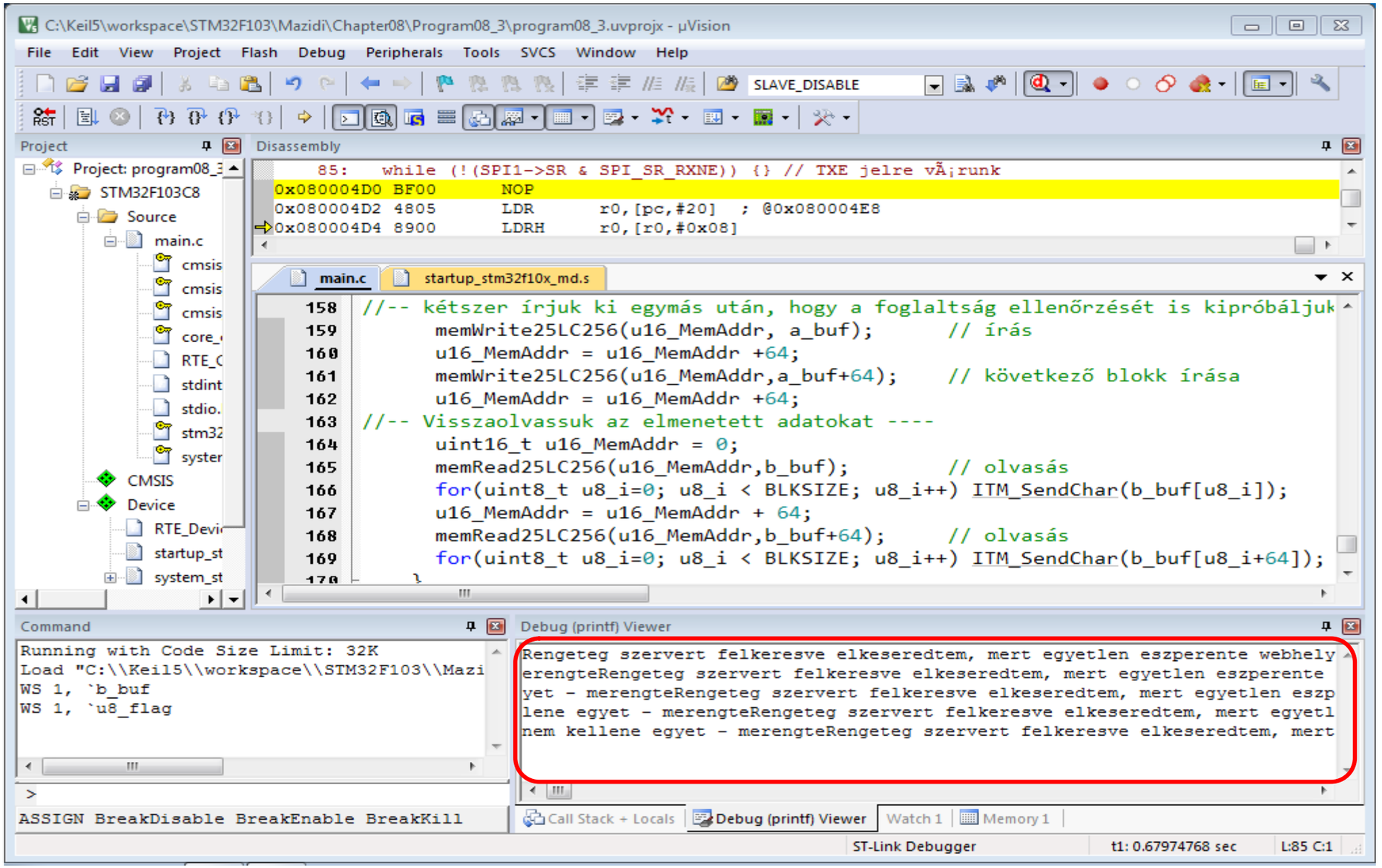
```
/*-------------------------------------------------------
 * Egy memórialap (64 bájt) olvasása az EEPROM egy
 * megadott címétől kezdődően, és eltárolása az adatbufferbe.
 * \param u16_MemAddr a beolvasni kívánt memórialap kezdőcíme
 * \param *pu8_buf mutató az adatbuffer kezdetéhez
 *-------------------------------------------------------*/
void memRead25LC256(uint16_t u16_MemAddr, char *pu8_buf) {
  uint8_t u8_i, u8_AddrLo, u8_AddrHi;
  u8_AddrLo = u16_MemAddr & 0x00FF;
  u8_AddrHi = (u16_MemAddr >> 8);
  WaitFor25LC256();                          // Várunk, ha az EEPROM elfoglalt
  SPI1_ENABLE();
  ioMasterSPI1(CMD_READ);                    // Adatblokk olvasása parancs
  ioMasterSPI1(u8_AddrHi);                   // Kezdőcím megadása
  ioMasterSPI1(u8_AddrLo);
  for(u8_i=0; u8_i < BLKSIZE; u8_i++) {
    *pu8_buf++=ioMasterSPI1(0x00);           // Adatbájtok átvitele
  }
  SPI1_DISABLE();
}
```

Folytatás a következő oldalon ...

Program08_3/main.c 6/6. oldal

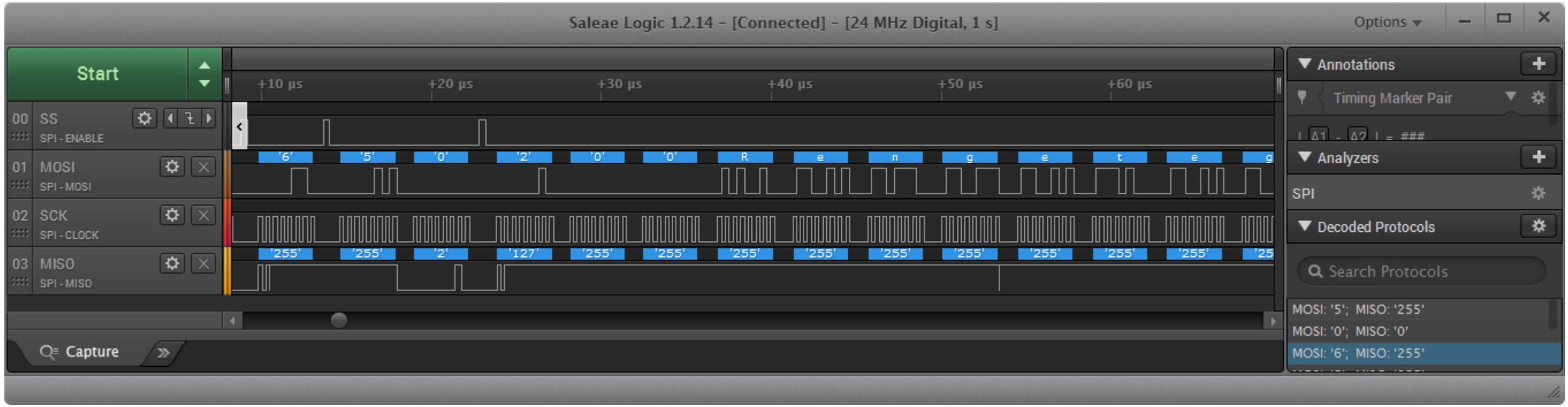
```
int main (void) {
  char a_buf[] = "Rengeteg szervert felkeresve elkeseredtem, mert egyetlen eszperente\
 webhelyet sem leltem. Szerkesztenem kellene egyet - merengtem egy este";
  char b_buf[BLKSIZE*2] = "";
  uint16_t u16_MemAddr;
  SPI1_init();                                  //az SPI1 csatorna konfigurálása
  u16_MemAddr = 0;                              //A memória 0 címétől kezdünk
  while (1) {
    //-- kétszer írjuk ki egymás után, hogy a foglaltság ellenőrzését is kipróbáljuk
    memWrite25LC256(u16_MemAddr, a_buf);         // írás
    u16_MemAddr = u16_MemAddr +64;
    memWrite25LC256(u16_MemAddr,a_buf+64);       // következő blokk írása
    u16_MemAddr = u16_MemAddr +64;
    //-- Visszaolvassuk és kiíratjuk az elmentett adatokat ----
    uint16_t u16_MemAddr = 0;
    memRead25LC256(u16_MemAddr,b_buf);           // olvasás
    for(uint8_t u8_i=0; u8_i < BLKSIZE; u8_i++) ITM_SendChar(b_buf[u8_i]);
    u16_MemAddr = u16_MemAddr + 64;
    memRead25LC256(u16_MemAddr,b_buf+64);        // olvasás
    for(uint8_t u8_i=0; u8_i < BLKSIZE; u8_i++) ITM_SendChar(b_buf[u8_i+64]);
  }
}
```

Program08_3 futási eredménye



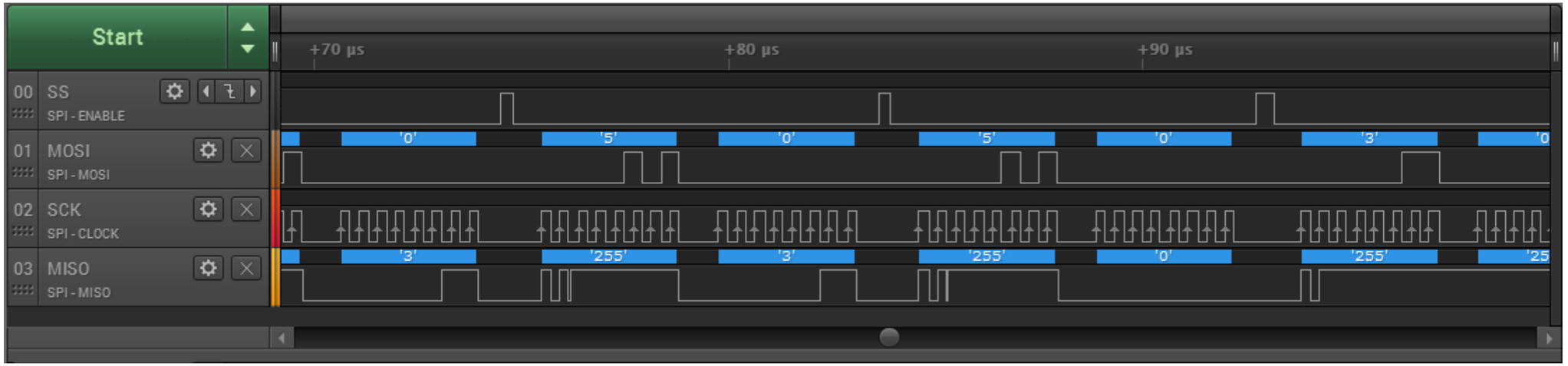
A programfutás ellenőrzése logikai analizátorral

- A képen az első adatblokk kiküldésével kapcsolatos parancsok láthatók:
- 6: **WREN** (írás engedélyezés) parancs, ami a **WEL** bitet 1-be állítja
- 5 0 : **RDSR** - foglaltság ellenőrzése. A kapott válasz 2, azaz **WEL**=1, **WIP**=0
- 2 0 0 : **WRITE** parancs: adatküldés kezdete és a memóriacím megadása
- R e n g ... : Az első 64 adatbájt küldése

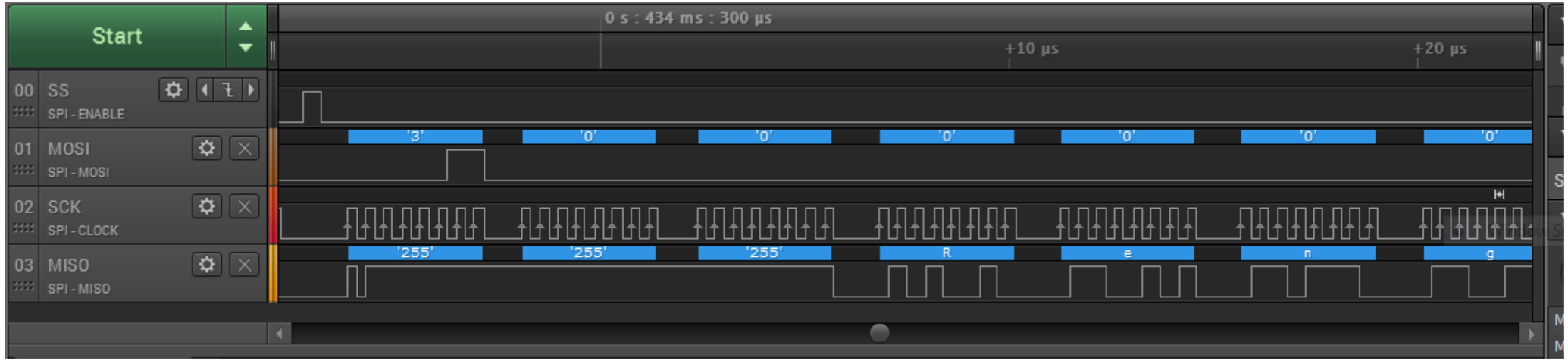


A programfutás ellenőrzése logikai analizátorral

- A felső képen a foglaltság vizsgálata (RDSR) ismétlődik (5 0, 5 0), amíg a válasz 0 nem lesz

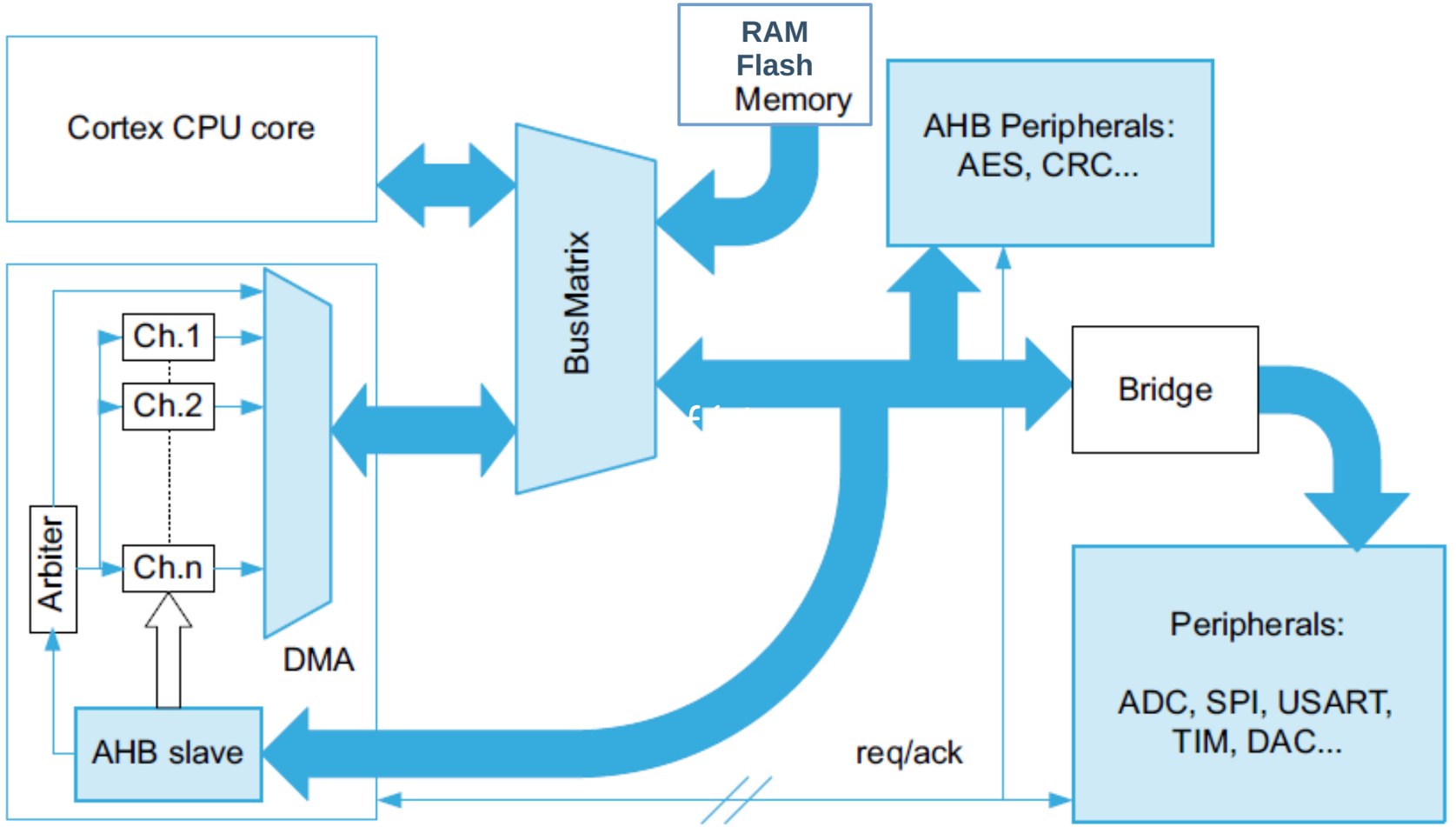


- Az alsó képen a blokk beolvasása kezdődik (3 0 0, majd a válasz: R e n g ...)



A DMA vezérlő használata

- A DMA vezérlő feladata, hogy a CPU-tól független adatmozgatásokat végezzen a memória és valamelyik periféria között, tehermentesítve a központi egységet
- **Főbb jellemzők:**
- 7 csatorna
- 8, 16, 32 bit szélesség
- automatikus címléptetés
- periféria – memória, ill. mem – mem műveletek
- 4 szintű prioritás
- 3 eseményjelző bit
- Max. 65 536 adat átvitele egy blokkban



DMA regiszterek

- **RCC_AHBENR** – az **AHB** perifériák órajelét engedélyező regiszter
  a megjelölt bitet 1-be kell állítani DMA1 engedélyezéséhez

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved					SDIO EN	Res.	FSMC EN	Res.	CRCE N	Res.	FLITF EN	Res.	SRAM EN	DMA2 EN	DMA1 EN
					rw		rw		rw		rw		rw	rw	rw

- **DMA1_ISR** – interupt status register (**TEIF*n*:** transfer error, **HTIF*n*:** half transfer interrupt
  TCIF*n*: Transfer complete interrupt, **GIF*n*:** channel global interrupt flag

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				TEIF7	HTIF7	TCIF7	GIF7	TEIF6	HTIF6	TCIF6	GIF6	TEIF5	HTIF5	TCIF5	GIF5
				r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TEIF4	HTIF4	TCIF4	GIF4	TEIF3	HTIF3	TCIF3	GIF3	TEIF2	HTIF2	TCIF2	GIF2	TEIF1	HTIF1	TCIF1	GIF1
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

- **DMA1_IFCR** – a fenti regiszter párja, melynek **valamely bitjébe <u>1-et írva</u>** törlődik a megfelelő jelzőbit

DMA csatornaregiszterek

- **DMA_CCR*n*** – csatorna konfigurációs regiszter (más jelöléssel **DMA1_Channel*n*->CCR**)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	MEM2MEM	PL[1:0]		MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR	TEIE	HTIE	TCIE	EN
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

- **MEM2MEM** – mem to mem mode, **PL** – priority level, **MSIZE** – mem size, **PSIZE** – periph. size, **MINC** – mem increment, **PINC** – Periph. Increment, **CIRC** – circular mode, **DIR** – data tarnsfer direction (**0:**P⟶M, **1:**M⟶P), **TEIE, HTIE, TCIE** - interrupt eneble, **EN** – channel enable

DMA_CNDTR*n*	DMA1_Channeln→CNDTR															NDT[15:0] Adatok száma															
Reset value																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DMA_CPAR*n*	DMA1_Channeln→CPAR															PA[31:0] Periféria cím															
Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DMA_CMAR*n*	DMA1_Channeln->CMAR															MA[31:0] Memória cím															
Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

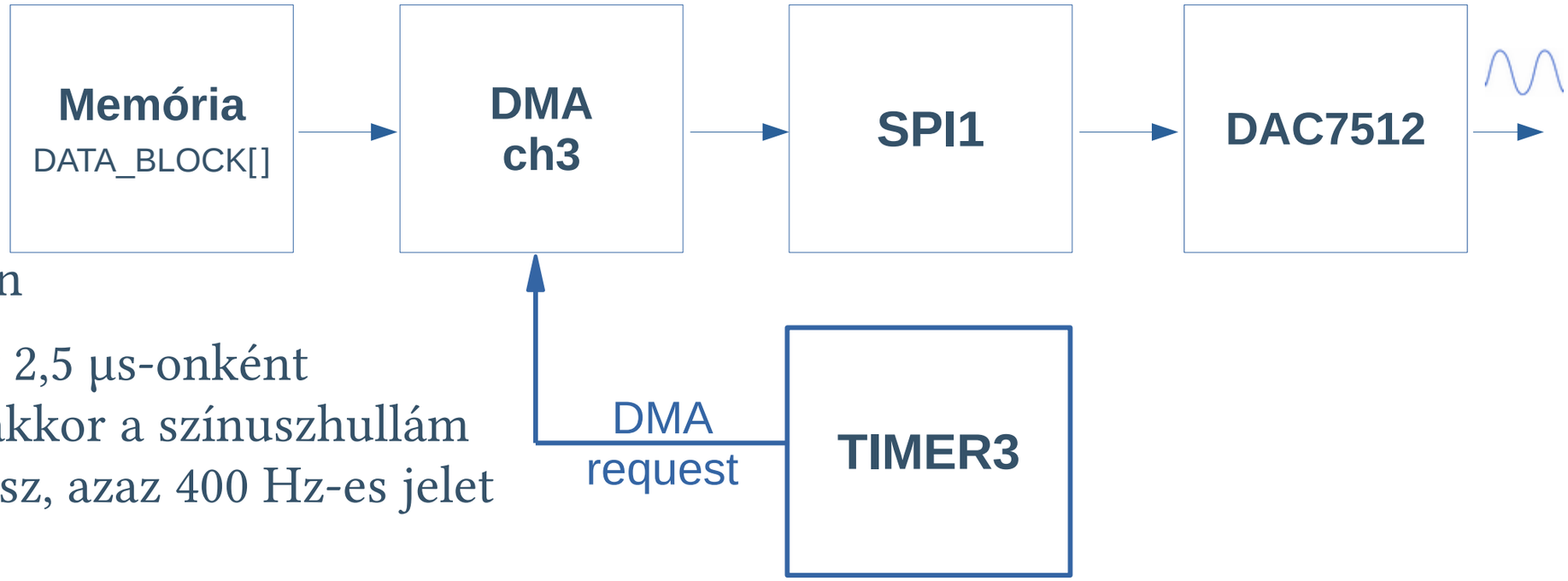
A DMA csatornák kiosztása

- Az alábbi táblázat azt foglalja össze, hogy melyik csatornán melyik periféria tud adatátvitelt kezdeményezni. A 8. előadásban már használtuk az 1. csatornát ADC-vel. Most a 3. csatornát fogjuk triggerelni a **TIM3_UP** eseménnyel

Peripherals	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
ADC1	ADC1	-	-	-	-	-	-
SPI/I^2S	-	SPI1_RX	SPI1_TX	SPI2/I2S2_RX	SPI2/I2S2_TX	-	-
USART	-	USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I^2C	-	-	-	I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	-	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP	-	-	TIM2_CH1	-	TIM2_CH2 TIM2_CH4
TIM3	-	TIM3_CH3	TIM3_CH4 TIM3_UP	-	-	TIM3_CH1 TIM3_TRIG	-
TIM4	TIM4_CH1	-	-	TIM4_CH2	TIM4_CH3	-	TIM4_UP

Program08_5: DAC vezérlés DMA átvitellel

- Ebben a példaprogramban az előző előadásban bemutatott **DAC7512** 12-bites digitális-analóg átalakítóra **DMA** segítségével az **SPI1** csatornán keresztül szeretnénk kiküldeni a memóriában előre letárolt adatokat (hullámtábla)
- Az adatok kiküldését ütemezetten, egy **Timer** segítségével végezzük, pl. **Timer3** Update eseményei indíthatják a **DMA** kérelmeket, a **DMA CH3** csatornája számára
- A memóriában a DATA_BLOCK[] nevű tömbben egy színusz-hullámot tárolunk 1000 adattal, ezt küldjük ki periodikusan
- Ha a **DMA** kérelmeket 2,5 µs-onként generáljuk (400 kHz), akkor a színuszhullám periódusideje 2,5 ms lesz, azaz 400 Hz-es jelet állítunk elő



Program08_5: DAC vezérlés DMA átvitellel

- Az alábbi **Python** nyelvű programmal kiszámoltathatjuk és egy C fejléc állományba menthetjük a szinuszhullám egy periódusát leíró adatokat:

  `python sinus.py > wt1000.h`

- Figyelembe véve, hogy 12 bites, nem negatív számokkal ábrázolunk, a nullát el kell tolni 2047-re, s az amplitúdó is 2047 lesz
- Ha n minta ír le egy periódust, akkor a szöglépés $2\pi/n$ lesz
- Az 1000 db adatot tartalmazó állományt **wt1000.h** néven hozzáadtuk a projekthez és a főprogram elején be is kell csatolni

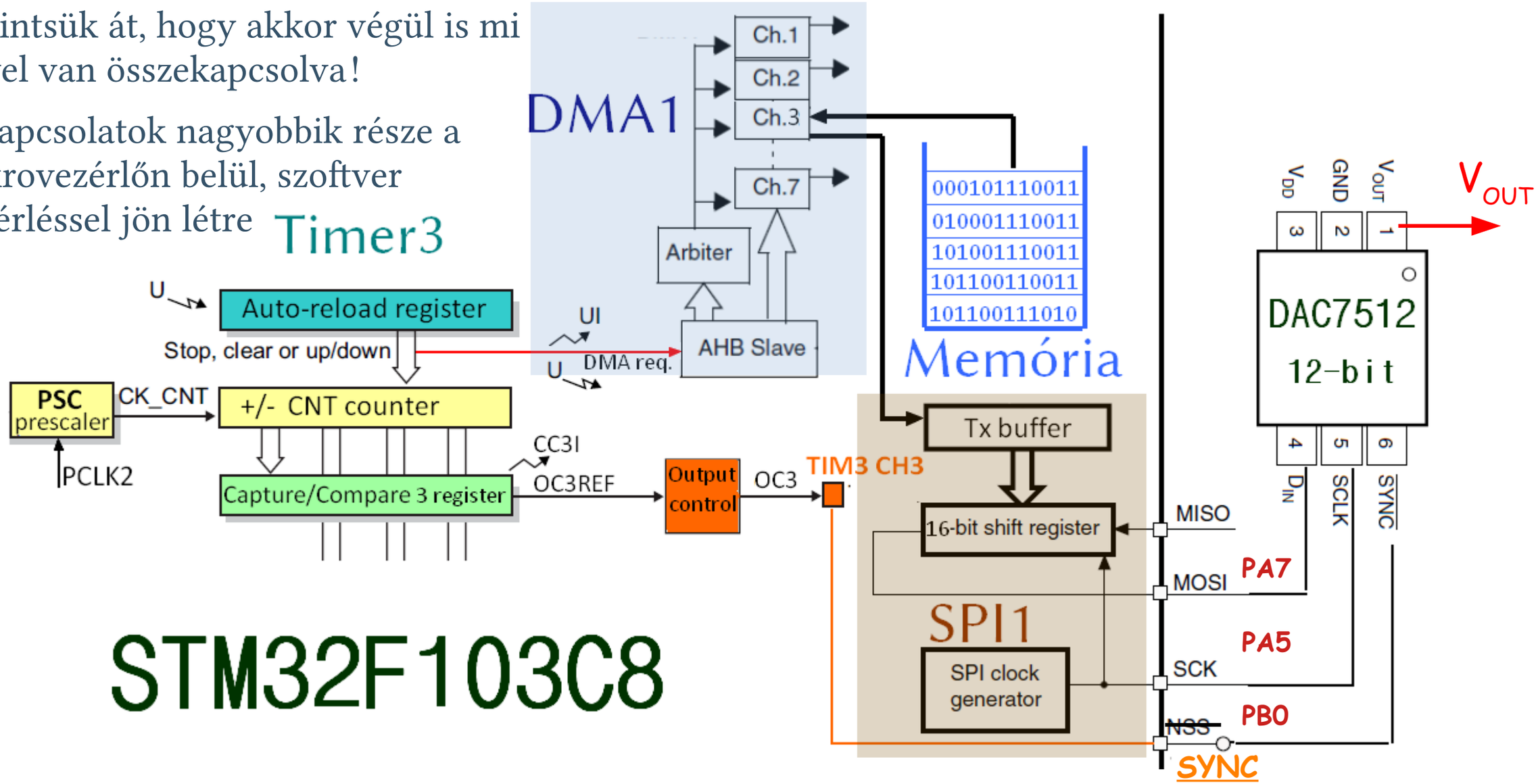
```
import math

def wave(n):
    for x in range(0,n):
        a = 2*x*math.pi/n
        y = 2047 + 2047*math.sin(a)
        if x%10 == 9 :
          term = "\n"
        else:
          term = " "
        print("%5d," % (y),end=term)

if __name__=='__main__':
   print("#ifndef BLOCK_SIZE")
   BLOCK_SIZE =  1000
   print("#define BLOCK_SIZE  %6d" % (BLOCK_SIZE))
   print("uint16_t DATA_BLOCK[] = {")
   wave(BLOCK_SIZE);
   print("};")
   print("#endif")
```

Porszem került a gépezetbe… de megoldjuk!

- Az előző oldalakon vázolt elgondolás, sajnos, mégsem működőképes. Mi a baj?
- A baj az, hogy a **DMA** használatával elvesztettük a lehetőséget az **NSS** jel vezérlésére:
  - Szoftveresen nincs módunk beavatkozni (nem tudjuk, hogy mikor történik az átvitel)
  - A hardveres vezérlés csak aktiválni tudja az **NSS** jelet, deaktiválni nem tudja
  - A **DAC7512** pedig állandóan alacsony szintre állított **SYNC** bemenettel nem működik, az adatküldés csak akkor hatásos, ha előtte legalább *33 ns* időtartamra a **SYNC** bemenetet magas szintre állítjuk
- **Megoldás: Timer3** modul a **DMA** kérelmek mellett, **PWM1** módban egy negatív polaritású impulzust is állítson elő, ami az Update eseménnyel egy időben indul és addig tart, amíg a számláló a **TIM3→CCR3** regiszterbe írt számot el nem éri, ez lesz a **SYNC** jel
- **Időzítések:**
  - az **SPI1** csatorna órajele 18 MHz így egy 16 bites átvitel ideje < 1 µs
  - **Timer3** órajelét 8 MHz-re állítjuk, a számlálási periódus pedig 2.5 µs (**TIM3→ARR=20**)
  - **Timer3 CH3** csatornájának kimenő jele 1,5 µs-ig van aktív állapotban (**TIM3→CCR3=12**)

Program08_5: DAC vezérlés DMA átvitellel – a megoldás

- Tekintsük át, hogy akkor végül is mi mivel van összekapcsolva!
- A kapcsolatok nagyobbik része a mikrovezérlőn belül, szoftver vezérléssel jön létre



Program08_5/main.c 4/1. oldal

```c
#include "stm32f10x.h"
#include "wt1000.h"
void SPI1_init16(void);
void Timer3_init(void);
void DMA1_init(void);
void DMA1_ch3_setup(void);

/*  Főprogram: 400 Hz-es szinuszhullámot generálunk a DAC kimenetén
    Az adatokat hullámtáblából DMA-val küldjük ki az SPI1 csatornán  */
int main(void) {
  SPI1_init16();                                  // Az SPI1 modul konfigurálása
  Timer3_init();                                  // TIM3 CH3 konfigurálása SS-vezérkéshez
  DMA1_init();                                   // DMA1 inicializálása
  while(1) {
    DMA1_ch3_setup();
    while(!(DMA1->ISR & DMA_ISR_TCIF3));          // Átvitel végére várunk
    DMA1->IFCR |= DMA_IFCR_CGIF3;                 // Törli a 3. csatorna jelzöbitjeit
  }
}
```

Program08_5/main.c 4/2. oldal

```
/*-------------------------------------------------------------------
    Az SPI1 csatorna konfigurálása 16 bites módba, 18 MHz bitrátával
    Az alapértelmezett PA5=SCK, PA7=MOSI kivezetéseket használjuk
    A Slave Select jelet nem az SPI1 modul szolgáltatja!
 *-------------------------------------------------------------------*/
void SPI1_init16(void) {
  RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;        // SPI1 órajel engedélyezés
  RCC->APB2ENR |= RCC_APB2ENR_AFIOEN;        // AFIO órajel engedélyezés
  RCC->APB2ENR |=  RCC_APB2ENR_IOPAEN;       // GPIOA órajel engedélyezés
  AFIO->MAPR &= ~AFIO_MAPR_SPI1_REMAP;       // Törli SPI1 REMAP bitjét
  /* PA5, PA7 konfigurálása, mint SPI1 SCK és MOSI */
  GPIOA->CRL &= ~0xF0F00000;                 // CNF és MODE bitek törlése
  GPIOA->CRL |=  0xB0B00000;                 // CNF=10 MODE=11 (AltFunc, pushpull kimenet)
  SPI1->CR1 = SPI_CR1_DFF  |                 // 16 bit mód
              SPI_CR1_BR_0 |                 // Baud rate 72/4 = 18MHz
              SPI_CR1_MSTR;                  // Master mód beállítás
  SPI1->CR2 = 0;                             // Nincs NSS vezérlés
  SPI1->CR1 |= SPI_CR1_SPE;                  // SPI1 engedélyezés
}
```

Program08_5/main.c 4/3. oldal

```
/*-------------------------------------------------------------------------
    Timer3 konfigurálása DMA 3. csatorna triggereléséhez
 *-------------------------------------------------------------------------*/
void Timer3_init(void) {
  //-- Timer3 konfigurálása ----------------------
  RCC->APB1ENR |= RCC_APB1ENR_TIM3EN;          // TIM3 idözítö engedélyezése
  RCC->APB2ENR |= RCC_APB2ENR_IOPBEN;          // GPIOB órajel engedélyezése
  //-- PB0 konfigurálása, mint TIM3 CH3 kimenet ---
  GPIOB->CRL |=  0x0000000B;                   // PB0 CNF=10 MODE=11 (AltFunc, pushpull kimenet)
  GPIOB->CRL &= ~GPIO_CRL_CNF0_0;              // A CNF_0 bit törlése
  TIM3->PSC = SystemCoreClock/8000000-1;       // 8 MHz-re osztjuk le az órajelet
  TIM3->ARR = 20 - 1;                          // 20-ig számlálunk (2.5 us)
  TIM3->CNT = 0;                               // Számláló nullázása
  TIM3->CCMR2 = 0x0060;                        // CH3 PWM1 mód
  TIM3->CCER  = TIM_CCER_CC3P |                // CH3 fordított polaritású
                TIM_CCER_CC3E;                 // CH3 engedélyezés
  TIM3->CCR3  = 12-1;                          // kb. 1,5 us az SS jel hossza
  TIM3->EGR  |= TIM_EGR_UG;                    // Regiszterek frissítése
  TIM3->CR1  |= TIM_CR1_CEN;                   // Számlálás engedélyezése
  TIM3->DIER |= TIM_DIER_TDE |                 // DMA kérések engedélyezése
                TIM_DIER_UDE;                  // TIM3 Update DMA kérés engedélyezése
}
```

A TIMER update eseményeket korábban már használtuk megszakítások generálására
Lásd: Program05_5

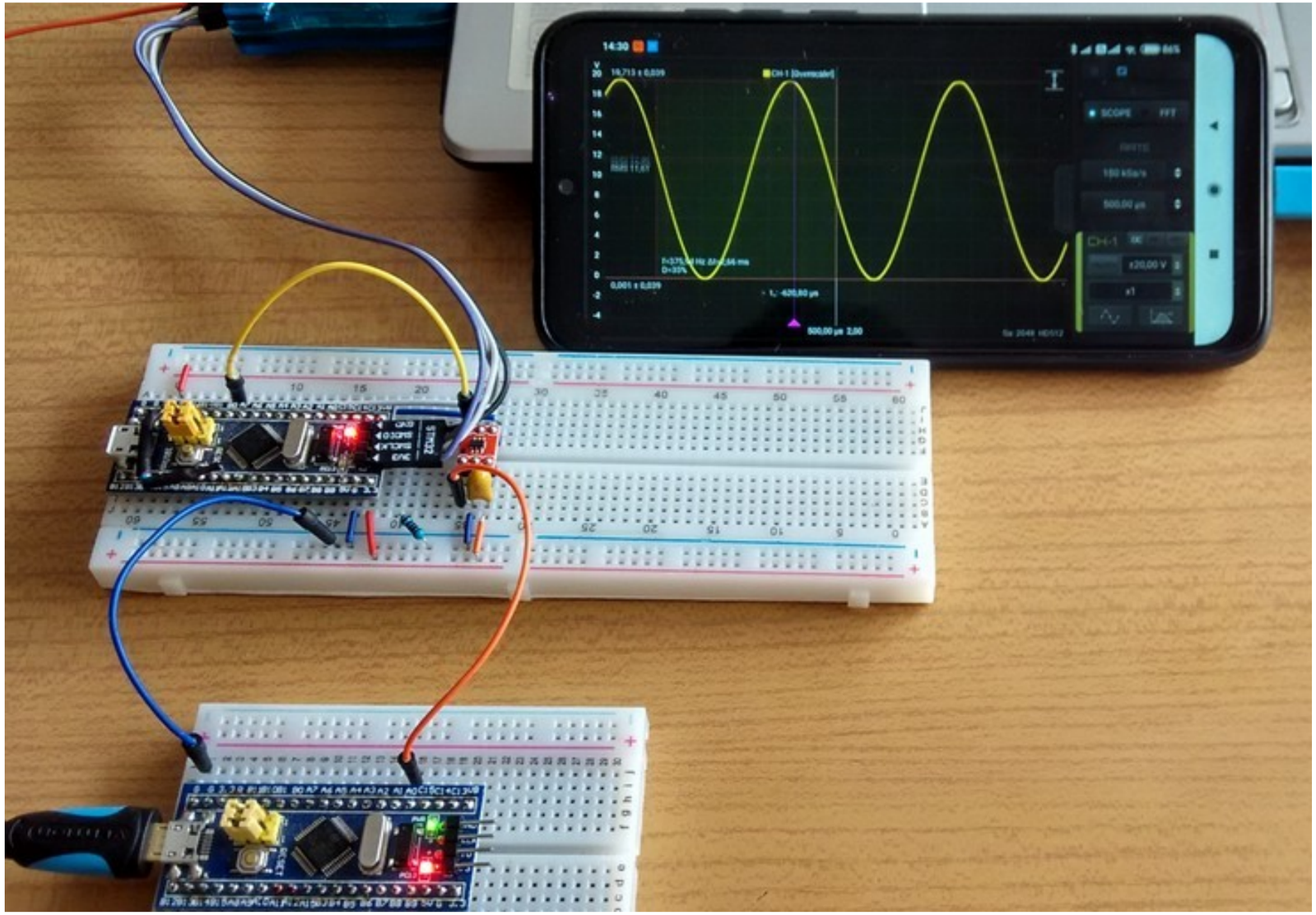
PWM1 ún. edge-aligned jelet kelt, ami az Update eseménykor lesz aktív, és a CCR regiszterben megadott szám elérésekor lesz inaktív

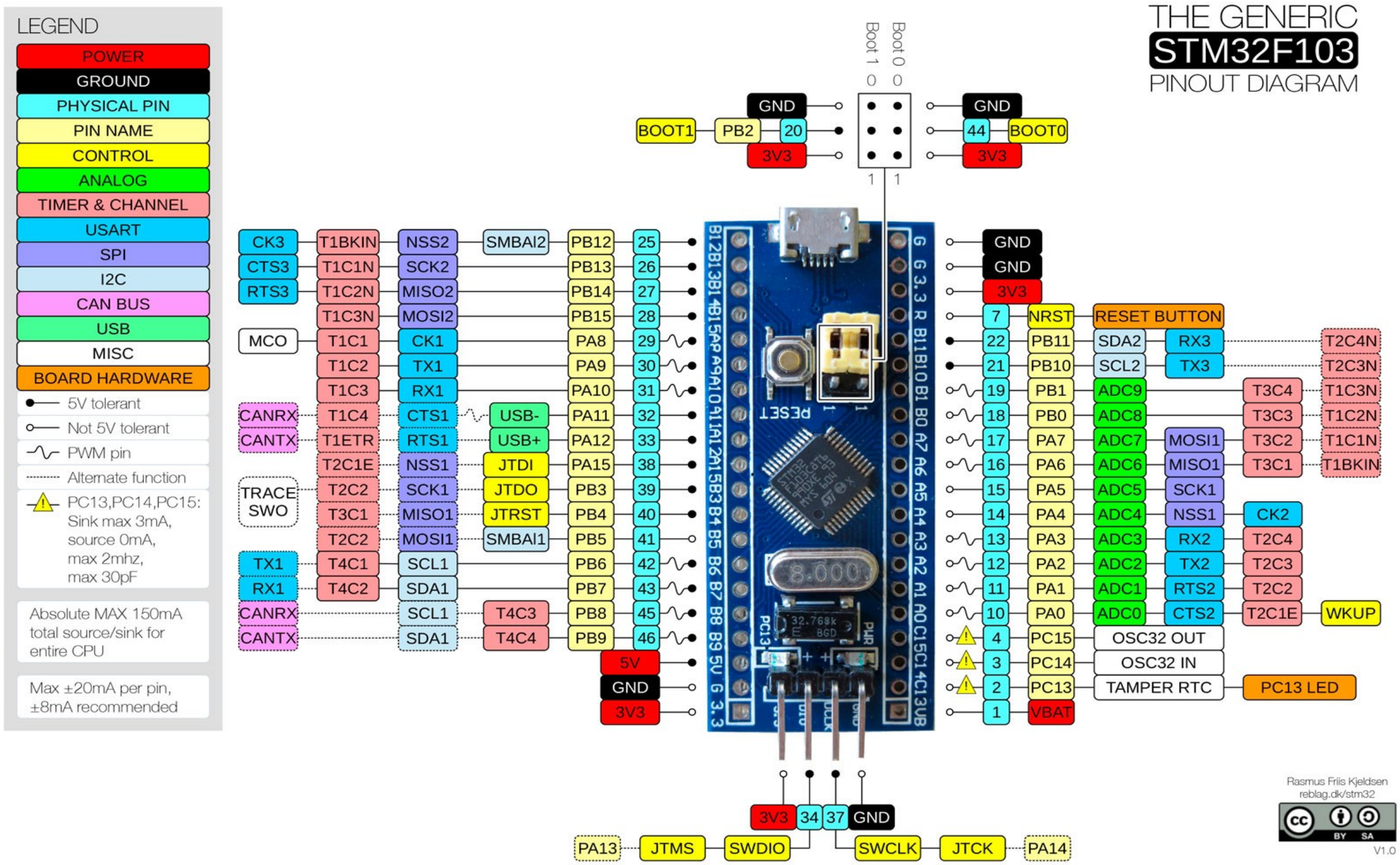
Program08_5/main.c 4/4. oldal

```
/*-----------------------------------------------------------------------
   A DMA1 vezérlő inicializálása
 *-----------------------------------------------------------------------*/
void DMA1_init(void) {
  RCC->AHBENR |= RCC_AHBENR_DMA1EN;          // DMA vezérlő engedélyezése
  DMA1->IFCR = 0x0FFFFFFF;                   // Törli a megszakítási jelzőbiteket
}

void DMA1_ch3_setup(void) {
  DMA1_Channel3->CCR = 0;                    // DMA1 Channel 3 ideiglenes letiltása
  while (DMA1_Channel3->CCR & 1) {}          // Vár, amíg DMA1 Channel 3 aktív
  DMA1->IFCR |= DMA_IFCR_CGIF3;              // Törli a 3. csatorna jelzőbitjeit
  DMA1_Channel3->CPAR = (int)&SPI1->DR;      // Peripheral  address  (cél)
  DMA1_Channel3->CMAR = (int)DATA_BLOCK;     // Memory address  (forrás)
  DMA1_Channel3->CNDTR = BLOCK_SIZE;         // length (adatblokk hossza)
  DMA1_Channel3->CCR = DMA_CCR3_MSIZE_0 |    // Memory size = 16
                       DMA_CCR3_PSIZE_0 |    // Peripheral size = 16
                       DMA_CCR3_MINC    |    // Memory increment mode enable
                       DMA_CCR1_DIR     |    // Write to peripherial
                       DMA_CCR3_PL_1;        // Priority level = High
  // DMA_CCR1_PINC = 0                       // Peripheral increment mode disabled
  DMA1_Channel3->CCR |= DMA_CCR3_EN;         // DMA1 Channel 3 engedélyezése
}
```

A program eredményének vizsgálata

- A **DAC7512** kimenő jelét az előző előadásban ismertetett **HScope** alkalmazás és egy **STM32** kártyára töltött **HS10x** firmware segítségével végeztük
- A **Hscope** alkalmazás hivatalos honlapja: hscope.martinloren.com/
- A **HS10x** firmware lelőhelye: github.com/martinloren/HScope




LEGEND
POWER
GROUND
PHYSICAL PIN
PIN NAME
CONTROL
ANALOG
TIMER & CHANNEL
USART
SPI
I2C
CAN BUS
USB
MISC
BOARD HARDWARE
5V tolerant
Not 5V tolerant
PWM pin
Alternate function
PC13,PC14,PC15: Sink max 3mA, source 0mA, max 2mhz, max 30pF
Absolute MAX 150mA total source/sink for entire CPU
Max ±20mA per pin, ±8mA recommended
THE GENERIC
STM32F103
PINOUT DIAGRAM
Boot 1
Boot 0
GND
BOOT1
PB2
20
3V3
GND
44
BOOT0
3V3
CK3
CTS3
RTS3
MCO
CANRX
CANTX
TRACE SWO
TX1
RX1
CANRX
CANTX
T1BKIN
T1C1N
T1C2N
T1C3N
T1C1
T1C2
T1C3
T1C4
T1ETR
T2C1E
T2C2
T3C1
T2C2
T4C1
T4C2
NSS2
SCK2
MISO2
MOSI2
CK1
TX1
RX1
CTS1
RTS1
NSS1
SCK1
MISO1
MOSI1
SCL1
SDA1
SCL1
SDA1
SMBAI2
USB-
USB+
JTDI
JTDO
JTRST
SMBAI1
T4C3
T4C4
PB12
PB13
PB14
PB15
PA8
PA9
PA10
PA11
PA12
PA15
PB3
PB4
PB5
PB6
PB7
PB8
PB9
5V
GND
3V3
25
26
27
28
29
30
31
32
33
38
39
40
41
42
43
45
46
GND
GND
3V3
7
22
21
19
18
17
16
15
14
13
12
11
10
4
3
2
1
NRST
PB11
PB10
PB1
PB0
PA7
PA6
PA5
PA4
PA3
PA2
PA1
PA0
PC15
PC14
PC13
VBAT
RESET BUTTON
SDA2
SCL2
ADC9
ADC8
ADC7
ADC6
ADC5
ADC4
ADC3
ADC2
ADC1
ADC0
OSC32 OUT
OSC32 IN
TAMPER RTC
RX3
TX3
MOSI1
MISO1
SCK1
NSS1
RX2
TX2
RTS2
CTS2
T3C4
T3C3
T3C2
T3C1
CK2
T2C4
T2C3
T2C2
T2C1E
PC13 LED
T2C4N
T2C3N
T1C3N
T1C2N
T1C1N
T1BKIN
WKUP
3V3
34
37
GND
PA13
JTMS
SWDIO
SWCLK
JTCK
PA14
Rasmus Friis Kjeldsen
reblag.dk/stm32
V1.0