Flash Programming of CC253x/4x devices

By Åsmund B. Bø

Keywords
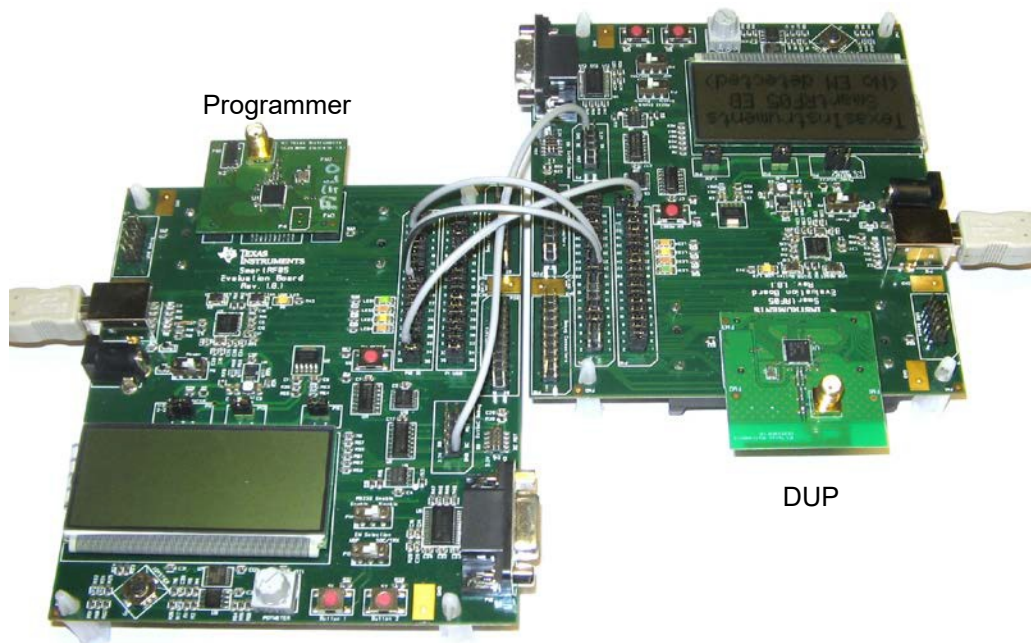
- *CC253x*
- *CC254x*
- *On-chip flash*
- *Flash programming*

- *Write flash*
- *Read flash*
- *Erase flash*
- *Debug interface*

1 Introduction

This application note demonstrates how to use the debug interface to perform various flash related tasks on a *CC253x* / *CC254x* System-on-Chip (SoC). This document is intended to help understanding the example code it is distributed along with [1]. The device being programmed is in this document referred to as DUP (Device Under Programming). The device programming the DUP is referred to as the "programmer". The example code is written for *CC2530* as a

programmer, but can easily be ported to other Low Power RF 8051 architecture SoCs. It covers how to enter debug mode and how to read/write/erase the flash memory. It does not show all the capabilities of the CC debug interface.

The example code is built using IAR Embedded Workbench for 8051, version 8.11.2. Project collateral discussed in this application note can be downloaded from the following URL: http://www.ti.com/lit/zip/SWRA410.

Programmer

DUP

Table of Contents

Table of Figures

2   Abbreviations

DC   Debug Clock

DD   Debug Data

DUP   Device Under Programming

EB   Evaluation Board

EM   Evaluation Module

LSB   Least Significant Byte

LSb   Least Siginficant bit

LPRF   Low Power RF

MSB   Most Significant Byte

MSb   Most Significant bit

SoC   System-on-Chip

3 Debug interface

The debug interface implements a proprietary two-wire serial interface that is used for in-circuit debugging. The interface allows programming of the on-chip flash, and it provides access to memory and register contents, in addition to features such as breakpoints, single-stepping, and register modification. The debug interface uses I/O pins P2.1 and P2.2 on the DUP, as debug data (DD) and debug clock (DC), respectively, during debug mode.

The DD pin is bi-directional, while DC is always controlled by the external host controller. Data is driven on the DD pin at the positive edge of the debug clock, and sampled on the negative edge of this clock. The idle state of the DC signal is logic 0. Please refer to the respective DUP's datasheet for debug interface timing requirements.

Debug commands are sent by an external host and consist of 1 to 4 output bytes from the host, and an optional input byte read by the host. The first byte of the debug command is a command byte. For more details about the debug interface, please refer to the *CC253x* / *CC254x* User's Guide [2]. A detailed description of the debug interface, also useful for *CC253x* / *CC254x* devices, is found in [3].

4 Hardware setup

The hardware assumed for the code example is what is found as a part of the *CC2530* development kit [4], but the code example supports any *CC253x* / *CC254x* LPRF 8051 device as a DUP:

- 2x SmartRF05 Evaluation Boards (SmartRF05EB)
- 2x *CC2530* Evaluation Modules (CC2530EM)

The necessary data lines for programming an LPRF 8051 architecture SoC, in addition to power and a common ground with the programmer, are shown in Figure 1. The jumper configuration and signal strapping between the programmer's and DUP's SmartRF05EB is shown in Figure 2.
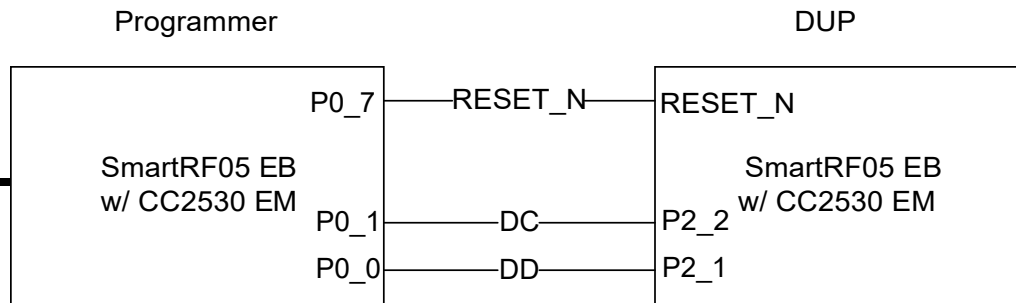
Figure 1 – Hardware connection between programmer and DUP
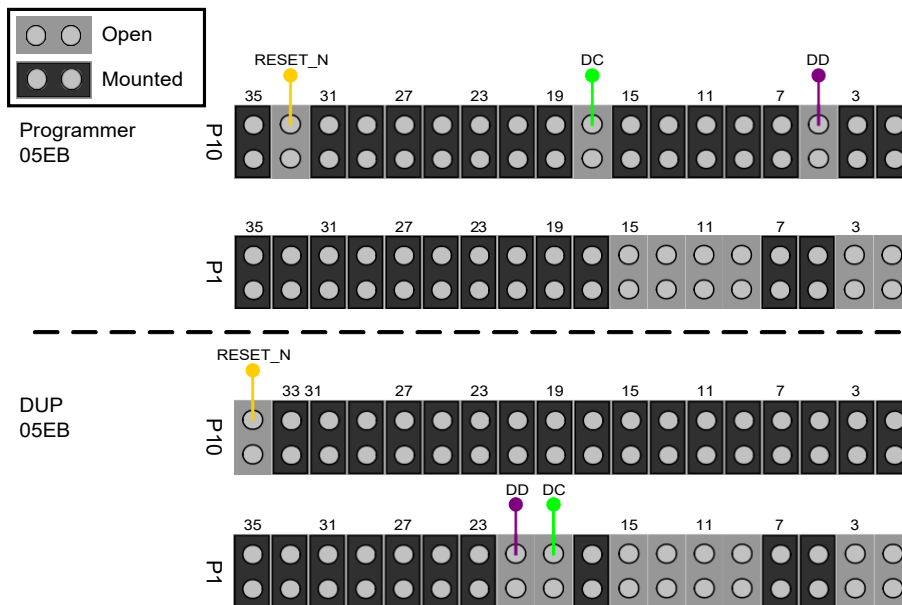
Figure 2 – SmartRF05EB P1/P10 jumper configuration for Programmer and DUP

The coloring in figures containing signalling in this document is as follows; the top signal (orange) is RESET_N, the middle signal (green) is Debug Clock (DC) and the bottom signal (purple) is Debug Data (DD).

5 Entering debug mode – debug_init()

Entering debug mode on CC devices is done by performing the following sequence on the DUT's RESET_N and DC line (P2.2).

1) Pull RESET_N low
2) Toggle two negative flanks on the DC line
3) Pull RESET_N high

The DUP is now in debug mode. The above sequence is shown in Figure 3 and Figure 4.

Figure 3 – Sequence on RESET_N and DC lines to enter debug mode
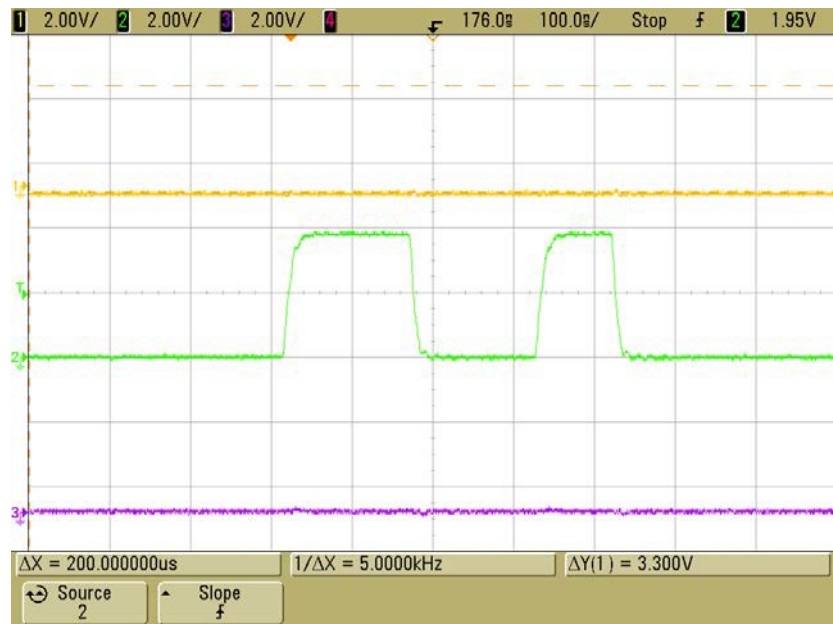
Figure 4 – Two negative flanks on DC line

6   Reading chip ID – read_chip_id()

After resetting a device into debug mode, one should first read its chip ID. Reading the device's chip ID can be done using the `GET_CHIP_ID()` debug instruction which returns the chip ID and chip revision (8+8 bits). The signalling sequence is shown in Figure 5.
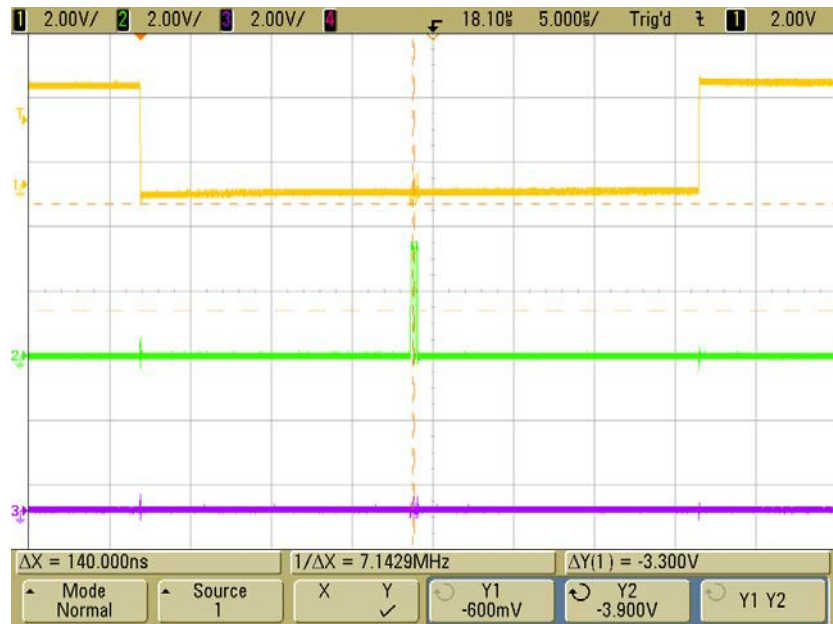
An overview of some of the CC device chip IDs is found in Table 1. The chip ID value returned by a device can also be found in the device datasheet.
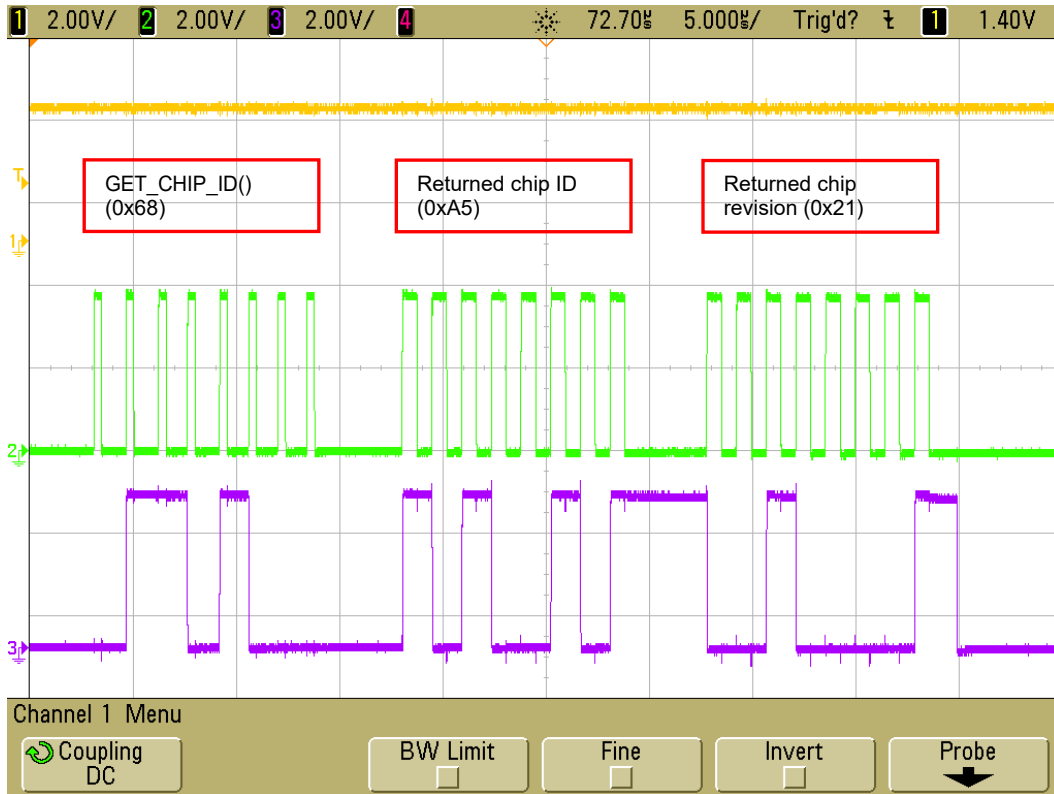
Figure 5 – GET_CHIP_ID() debug instruction

Chip	Chip ID	Page erase size	SRAM size
CC2530	0xA5	2 KB	8 KB
CC2531	0xB5	2 KB	8 KB
CC2533	0x95	1 KB	4 KB / 6 KB
CC2543	0x43	1 KB	1 KB
CC2544	0x44	1 KB	2 KB
CC2545	0x45	1 KB	1 KB

Table 1 – Chip ID and page erase size for a selection of CC devices

7 Erasing FLASH memory – chip_erase()

All debug interface activity must be performed after resetting the chip in debug mode. When the DUP is in debug mode, the DUP's entire flash memory can be erased by using the `CHIP_ERASE()` debug instruction, as seen in Figure 6. More details can be found in [2] and [3].

If a `CHIP_ERASE()` is performed prior to e.g. programming a device, one should wait until the chip erase has completed. Bit 7 of the returned data after the `READ_STATUS()` debug instruction is used to check this.

It is also possible to erase a single page by using the `DEBUG_INSTR()` debug instruction. The page erase size for different DUPs are given in Table 1 on page 7. The sequence for erasing a single flash page is given below. More details about erasing a single flash page can be found in section 6.3 of [2].

1. Point Flash controller to page's start address (`FADDRH[6:0]` or `FADDRH[7:1]`, see [2])
2. Trigger flash controller to start (set `FCTL.ERASE = 1`)
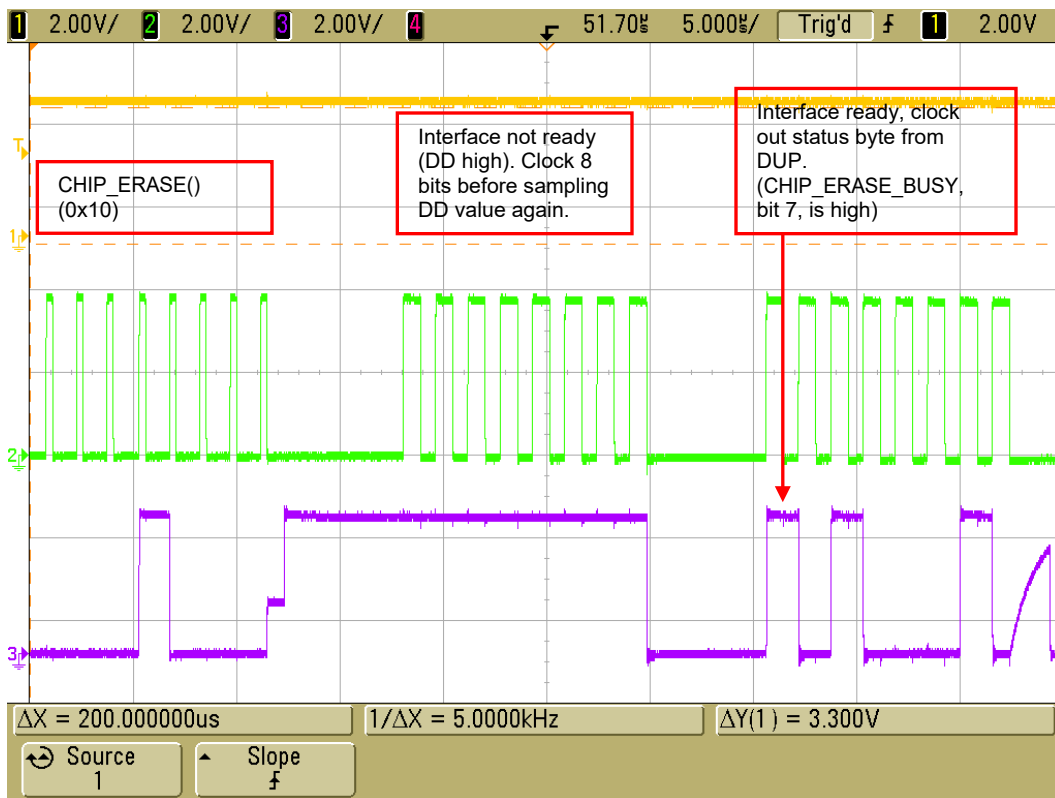3. Wait for page erase to complete (poll `FCTL.BUSY`)

Figure 6 – CHIP_ERASE() debug instruction

8   Writing to FLASH memory – write_flash_memory_block()

There are two ways to write the flash memory on *CC253x* / *CC254x* devices, either by using DMA transfer (preferred), or by using the CPU, running code from SRAM. There is no debug interface instruction to write data directly to the DUP's flash.

The preferred way to write flash is to configure two DMA channels on the DUP that a) read from the debug interface and b) feeds the DUP's Flash controller. To do this, we utilize the DUP's SRAM to store the DMA channel configurations (2 x 8 B) and an internal buffer. The size of the internal buffer is in the example code set to 512 B due to the *CC2543* and *CC2545* SRAM size (1 KB). A bigger internal buffer increases performance due to the reduced command overhead.

The sequence for writing flash memory using DMA channels on the DUP is as follows:

1. Enable use of DMA in debug configuration
2. Transfer DMA configuration to SRAM
3. Point DMA controller to DMA configuration
4. Point Flash controller to start address
5. Arm DMA channel that triggers on debug interface data (ch. 1)
6. Transfer data over debug interface (using BURST_WRITE())
7. Arm DMA channel that feeds Flash controller (ch. 2)
8. Trigger flash controller to start

Details on each of the above steps are given in the following sections.

8.1   Enable use of DMA in debug configuration

To enable use of DMA, the PAUSE_DMA bit in the DUP's debug configuration is cleared using the `WR_CONFIG()` debug instruction. Figure 7 shows the corresponding debug interface traffic.
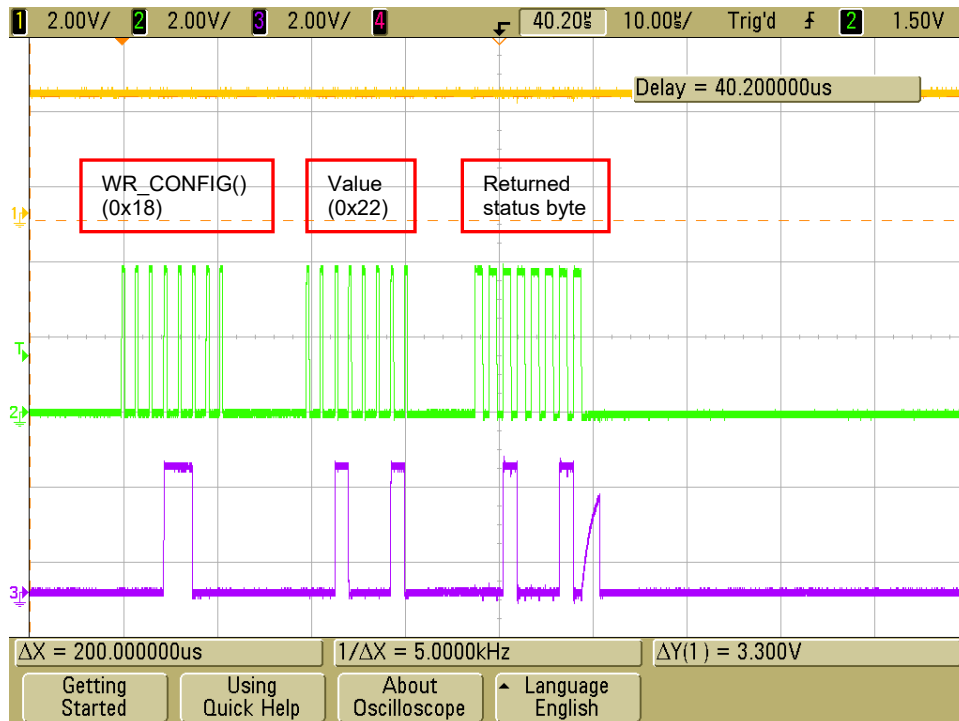

Figure 7 – Debug instruction WR_CONFIG() writing 0x22 to DUP

8.2 DMA configurations

One DMA channel is configured to transfer data from the debug interface to a buffer in SRAM, while the second transfers data from the internal buffer to the Flash controller. In this example, we make a 512 B SRAM buffer at address 0x0000, and place the DMA configurations at address 0x0200. Each DMA configuration consists of 8 B.

- 0x0000 (SRAM)   512 byte buffer for data sequence (BUF0)
- 0x0200 (SRAM)   8 byte DMA descriptor
- 0x0208 (SRAM)   8 byte DMA descriptor

The minimum amount of data to be written by the Flash controller is 32 bit (4 B). Note that caution should be made related to multiple writes to flash memory without prior CHIP_ERASE(). This is described in section 6.2.2 of [2]. In the same document, you'll find more details on the DMA configuration structure for **CC253x** / **CC254x** devices. Any DMA channel on the DUP can be used, the example code uses DMA channel 0 and 1.

The DMA configurations (8+8 B) are transferred to the DUP's SRAM by using the DEBUG_INSTR() instruction.

8.2.1   DMA channel 0: Debug interface to Internal SRAM buffer

Byte offset	Bit	Name	Value	Description
0	7:0	SRCADDR[15:8]	0x62	DBGDATA register address (MSB)
1	7:0	SRCADDR[7:0]	0x60	DBGDATA register address (LSB)
2	7:0	DESTADDR[15:8]	0x00	BUF0 start address, 0x0000 (MSB)
3	7:0	DESTADDR[7:0]	0x00	BUF0 start address, 0x0000 (LSB)
4	7:5	VLEN[2:0]	000b	Use LEN for transfer count
4	4:0	LEN[12:8]	–	DMA channel transfer count (MSB)
5	7:0	LEN[7:0]	–	DMA channel transfer count (LSB)
6	7	WORDSIZE	0	Each DMA transfer is 8 bit
6	6:5	TMODE[1:0]	00b	Single mode DMA transfer
6	4:0	TRIG[4:0]	0x1F	DBG_BW trigger
7	7:6	SRCINC[1:0]	00b	Do not increment source address
7	5:4	DESTINC[1:0]	01b	Increment destination 1 B/word.
7	3	IRQMASK	0	Disable interrupt generation
7	2	M8	0	Use all 8 bits for transfer count
7	1:0	PRIORITY[1:0]	01b	Assured, DMA at least every second try

Table 2 – Configuration for DMA channel 0

8.2.2 DMA channel 1: Internal SRAM buffer to Flash Controller

Byte offset	Bit	Name	Value	Description
0	7:0	SRCADDR[15:8]	0x00	BUF0 start address, 0x0000 (MSB)
1	7:0	SRCADDR[7:0]	0x00	BUF0 start address, 0x0000 (LSB)
2	7:0	DESTADDR[15:8]	0x62	Flash controller's FWDATA address (MSB)
3	7:0	DESTADDR[7:0]	0x73	Flash controller's FWDATA address (MSB)
4	7:5	VLEN[2:0]	000b	Use LEN for transfer count
4	4:0	LEN[12:8]	–	DMA channel transfer count (MSB)
5	7:0	LEN[7:0]	–	DMA channel transfer count (LSB)
6	7	WORDSIZE	0	Each DMA transfer is 8 bit
6	6:5	TMODE[1:0]	00b	Single mode DMA transfer
6	4:0	TRIG[4:0]	0x12	FLASH trigger
7	7:6	SRCINC[1:0]	01b	Increment source address 1 B/word
7	5:4	DESTINC[1:0]	01b	No not increment destination address
7	3	IRQMASK	0	Disable interrupt generation
7	2	M8	0	Use all 8 bits for transfer count
7	1:0	PRIORITY[1:0]	10b	High, DMA has priority

Table 3 – Configuration for DMA channel 1

The DMA channel writing data to the DUP's Flash controller should have the highest priority to avoid data underflow to the flash controller.

8.3 Point DMA controller to DMA configurations

By using the DEBUG_INSTR() instruction, the SRAM start address of the DMA configurations are written to the DMAxCFGH:DMAxCFGL registers (x=0,1). By writing DMA0CFGy registers (y=H,L), we configure DMA channel 0 and by writing DMA1CFGy registers, we configure DMA channel 1. See [2] for details.

8.4 Arm DMA channel 0

The configured DMA channels (0 and 1) are armed by writing the corresponding bit in the DUP's DMAARM register. The DEBUG_INSTR() instruction is used for this. See Figure 11 on page 14 for screenshot of the sequence for arming DMA channel 1.

8.5 Point Flash controller to start address

By using the `DEBUG_INSTR()` instruction, the flash controller start address is set by writing DUP's registers `FADDRH:FADDRL`. These registers hold the 16 MSb of the 18 bit address. Figure 8 and Figure 9 show the corresponding debug interface traffic.
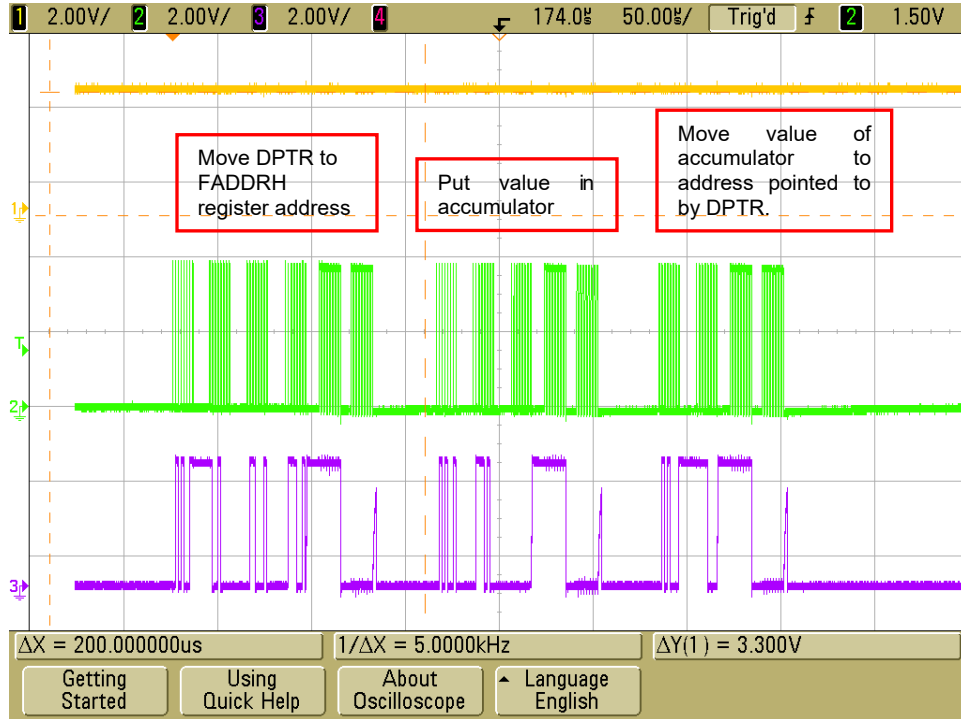

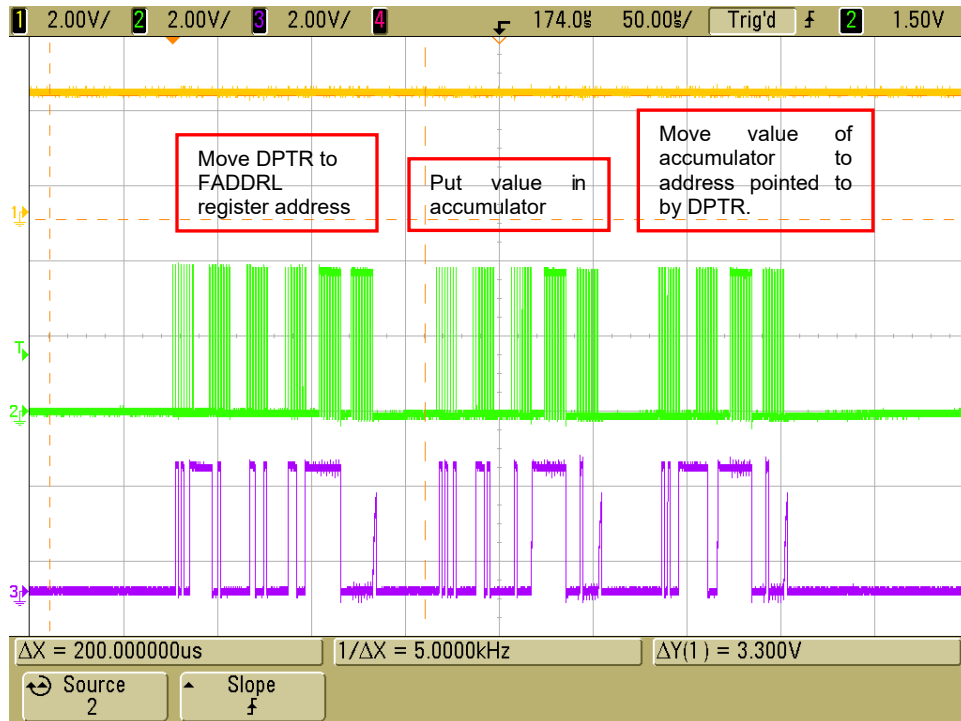**Figure 8 – Writing 8 MSb [17:10] of 18-bit flash start address to FADDRH**

Figure 9 – Writing bit [9:2] of flash start address to FADDRL

8.6 Transfer data using BURST_WRITE

The `BURST_WRITE()` instruction enables us to transfer 1-2048 B over the debug interface. DMA channel 0 on the DUP is now configured to transfer data from the DUP's `DBGDATA` register and place it in SRAM. It is therefore important that DMA transfers are enabled (section 8.1) and that the configured DMA channel is armed (section 8.4). Figure 10 shows the corresponding debug interface traffic.
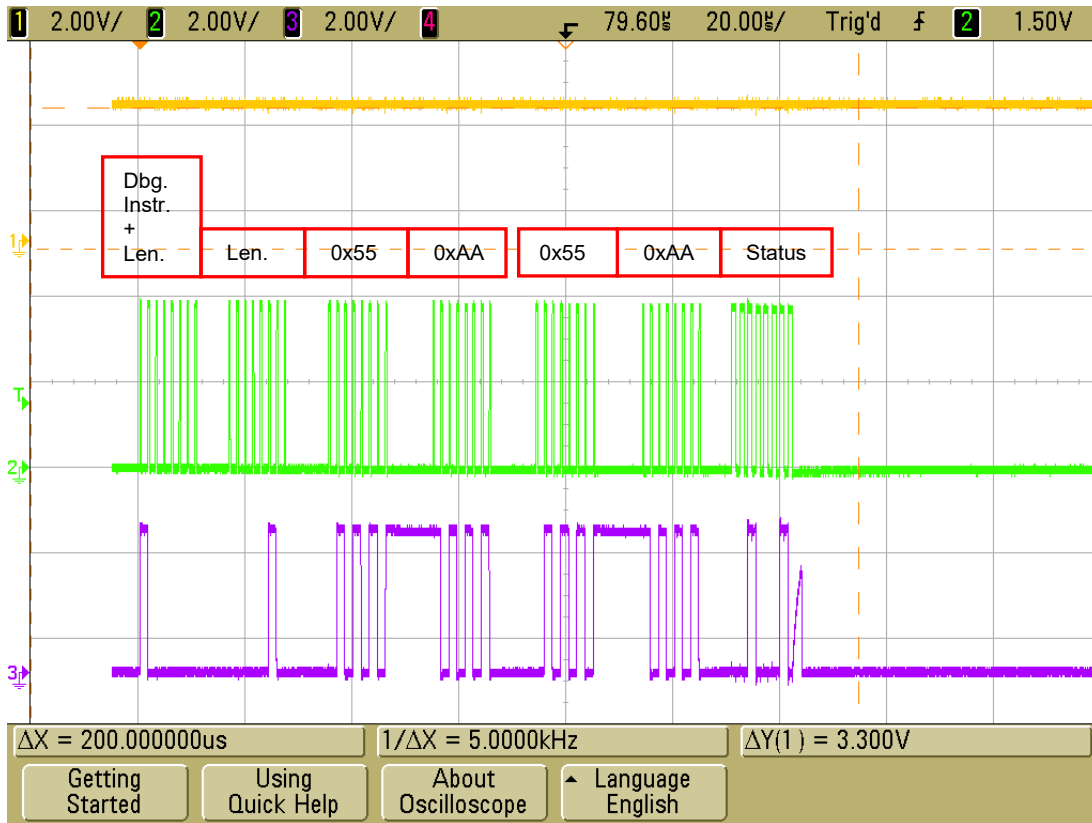
Figure 10 – BURST_WRITE() 4 bytes over the debug interface

8.7 Arm DMA channel 1

Now the data we want to transfer is stored in SRAM. DMA channel 1 is armed by setting the corresponding bit in the DUP's `DMAARM` register. The `DEBUG_INSTR()` instruction is used for this. Figure 11 shows the corresponding debug interface traffic.
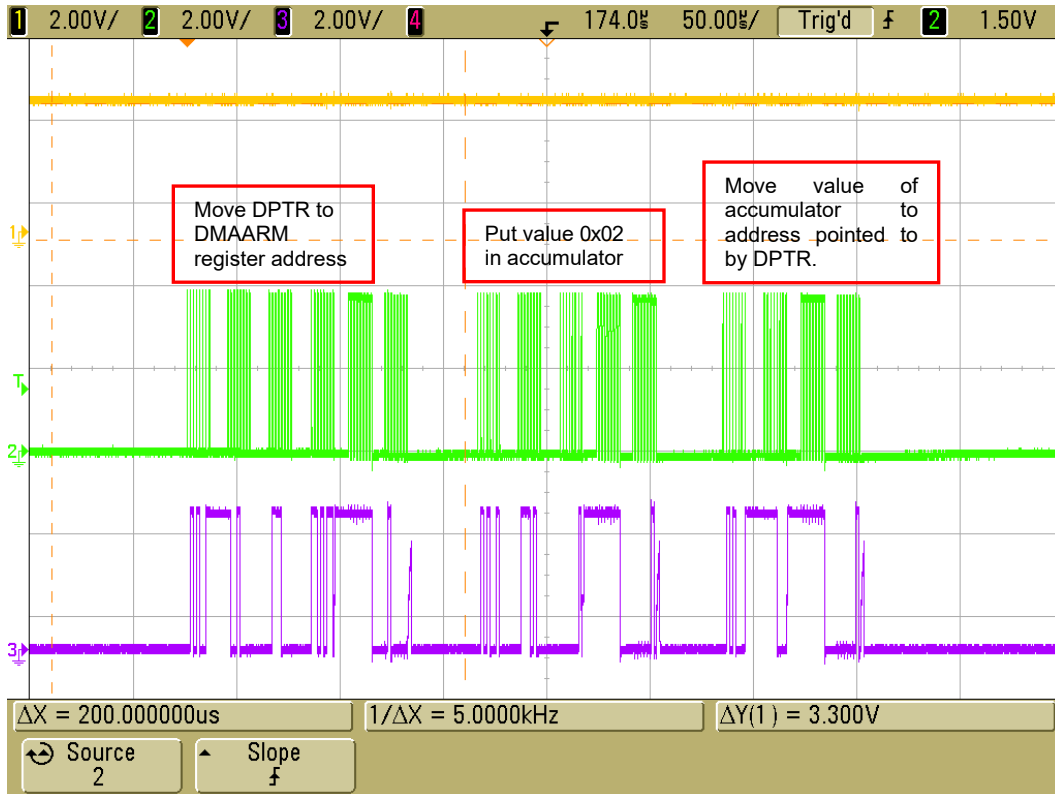
Figure 11 – Writing 0x02 to DUP's DMAARM register to arm DMA channel 1

8.8 Trigger flash controller

The flash controller flash write procedure is started by setting the `WRITE` bit in the DUP's `FCTL` register. This triggers the Flash controller, which in turn triggers the DMA channel that feeds the Flash controller. The `WRITE` bit is set by using the `DEBUG_INSTR()` debug instruction. Figure 12 shows the corresponding debug interface traffic.

The programming is completed when the Flash controller's status bit (`FCTL.BUSY`) returns to 0.
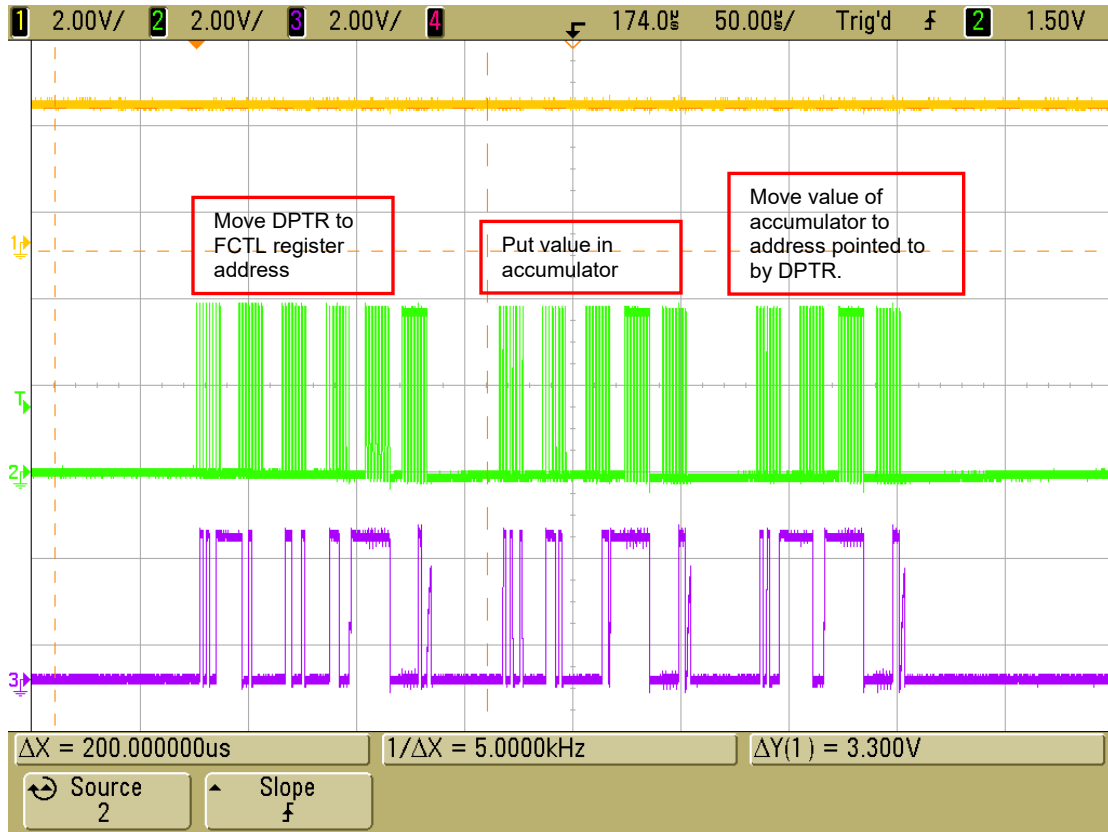
Figure 12 – Set FCTL.WRITE bit to 1 to initiate flash programming

9 Reading from FLASH memory – read_flash_memory_block()

To read data from the DUP's flash memory, the `DEBUG_INSTR()` instruction is used. It performs the CPU instructions given, and returns the value of the DUP's accumulator register after the issued instruction.

The sequence to read from flash memory is as follows:
1. Map flash memory bank to XDATA address 0x8000 – 0xFFFF
2. Move data pointer (DPTR) to 0x8000 + <flash memory block start address>
3. Move value pointed to by DPTR to accumulator register
4. Increment data pointer (DPTR)

Steps 3-4 are repeated for up to 32 KB, until a new flash memory bank must be mapped to XDATA memory space. Figure 13 shows an example of how to implement the above sequence. For more details on the used functions, see the source files of the code example. The DD and DC line activity corresponding to step 1-4 is shown in Figure 14 through Figure 17.

```c
void read_flash_memory_block(unsigned char bank,
                             unsigned short flash_addr,
                             unsigned short num_bytes,
                             unsigned char *values)
{
    unsigned char instr[3];
    unsigned short i;
    unsigned short xdata_addr = (0x8000 + flash_addr);

    // 1. Map flash memory bank to XDATA address 0x8000-0xFFFF
    write_xdata_memory(DUP_MEMCTR, bank);

    // 2. Move data pointer to XDATA address (MOV DPTR, xdata_addr)
    instr[0] = 0x90;
    instr[1] = HIBYTE(xdata_addr);
    instr[2] = LOBYTE(xdata_addr);
    debug_command(CMD_DEBUG_INSTR_3B, instr, 3);

    for (i = 0; i < num_bytes; i++)
    {
        // 3. Move value pointed to by DPTR to accumulator
        // (MOVX A, @DPTR)
        instr[0] = 0xE0;
        values[i] = debug_command(CMD_DEBUG_INSTR_1B, instr, 1);

        // 4. Increment data pointer (INC DPTR)
        instr[0] = 0xA3;
        debug_command(CMD_DEBUG_INSTR_1B, instr, 1);
    }
}
```
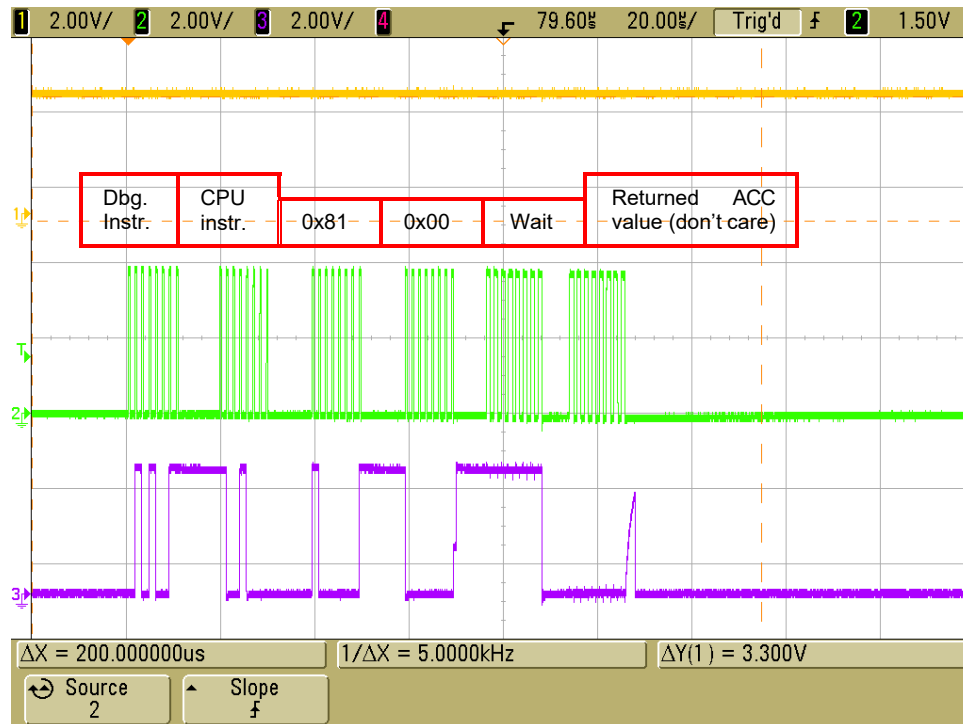
Figure 13 – Function example for reading flash memory via debug interface
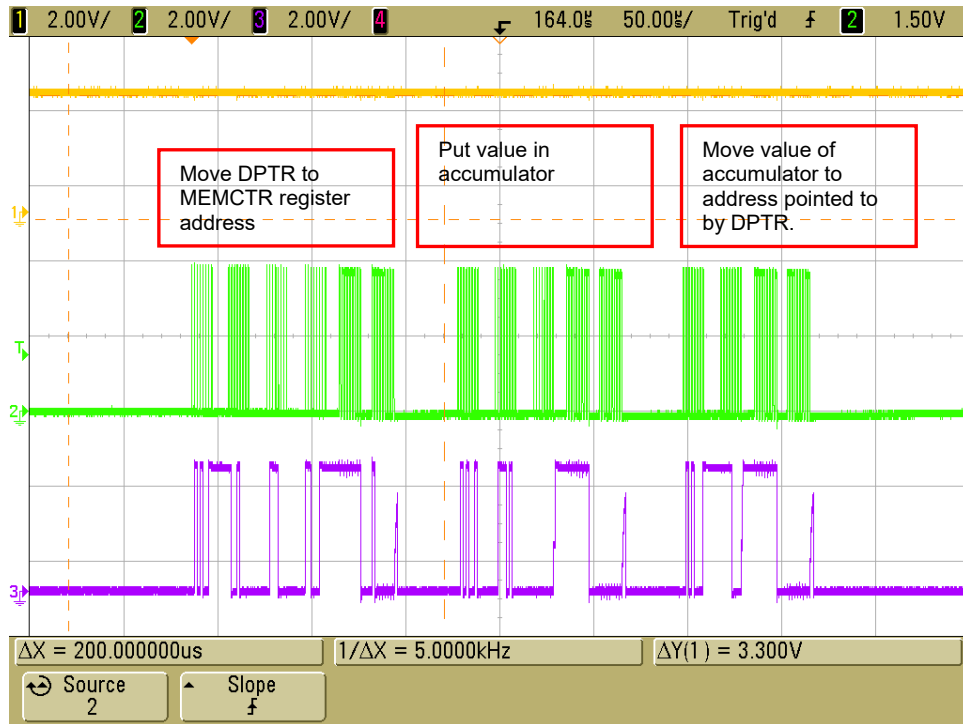
Figure 14 – Step 1: Map flash memory bank to XDATA memory space

The peaks seen on the DD line in e.g. Figure 14 are a result of the DD line direction transition, i.e. the direction is changed from being output (driven by programmer) to input (driven by DUP).

Figure 15 – Step 2: Move data pointer to start address (XDATA 0x8100 in this case, corresponding to flash memory address 0x0100)
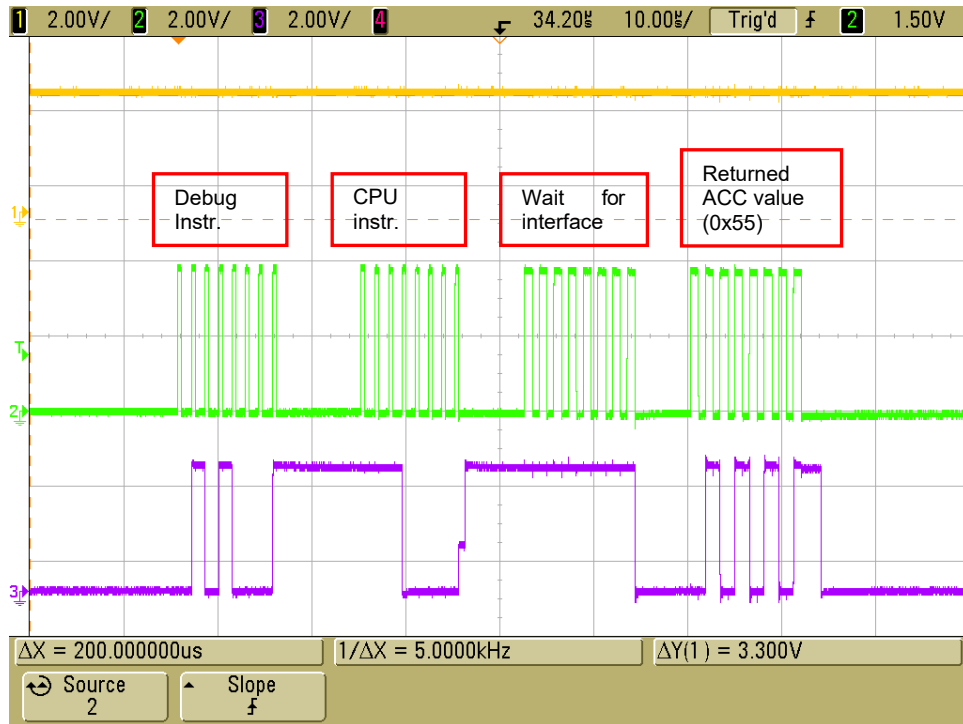
Figure 16 – Step 3: Move value at DPTR to accumulator (value of ACC is returned on the debug interface). Returned value is 0x55 (value of flash memory address 0x0100)
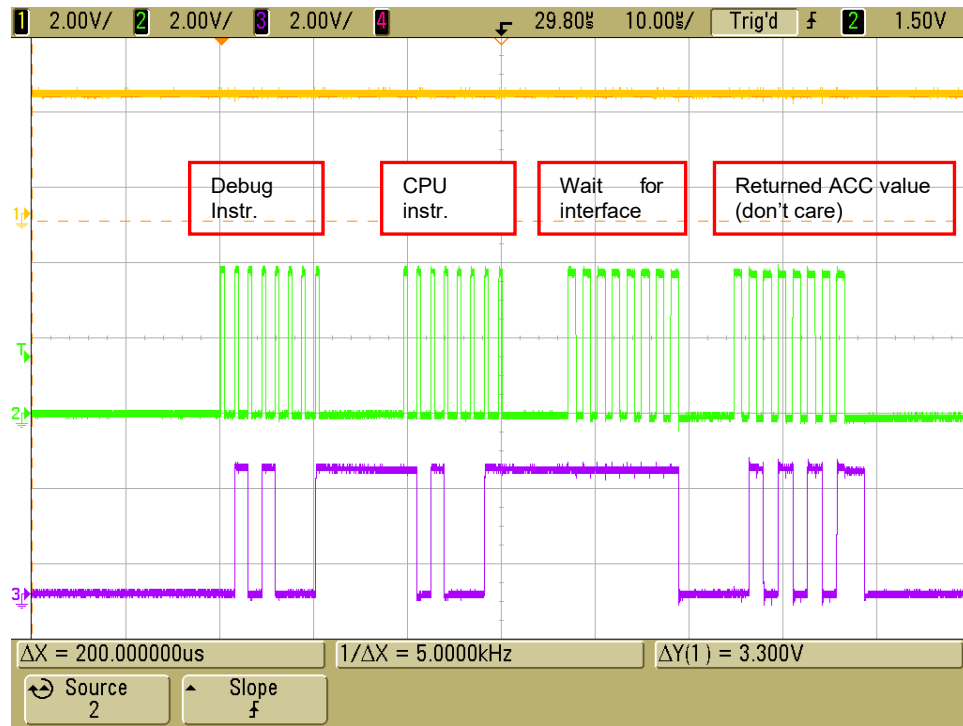
Figure 17 – Step 4: Increment DPTR

10 References

[1]   Flash programming of CC253x/4x devices code example
        http://www.ti.com/lit/zip/swra410

[2]   CC253x/4x User's Guide
        http://www.ti.com/lit/swru191

[3]   CC1110/CC2430/CC2510 Debug and Programming Interface Specification
        http://www.ti.com/lit/swra124

[4]   CC2530DK User's Guide
        http://www.ti.com/lit/swru208

11 Document history

Version	Date	Description/Changes
SWRA410	2012-09-04	Initial version.

IMPORTANT NOTICE AND DISCLAIMER