# User manual

## DA14580 Creation of a secondary boot loader

### UM-B-012

**Abstract**

*This document provides an overview of the booting sequence of the DA14580 and it describes the implementation steps for the development of a secondary boot loader application An extension of the secondary boot loader to support a dual boot images scheme is also presented.*

# Contents



# Figures



# Tables

# 1 Terms and definitions

| | |
|---|---|
| EEPROM | Electrically Erasable Programmable Memory |
| GPIO | General Purpose Input Output |
| OTP | One Time Programmable (memory) |
| SDK | Software Development Kit |
| SPI | Serial Peripheral Interface |
| SUOTA | Software Update Over The Air |
| UART | Universal Asynchronous Receiver/Transmitter |
| URX | UART Receive port |
| UTX | UART Transmit port |

# 2 References

1. DA14580 Datasheet, Dialog Semiconductor.
2. AN-B-001, Booting from Serial Interfaces, Dialog Semiconductor
3. UM-B-014, DA14580 Development Kit, Dialog Semiconductor.
4. UM-B-003, DA14580 Peripherals Drivers, Dialog Semiconductor.
5. UM-B-009, DA14580 Proximity Application, Dialog Semiconductor.
6. AN-B-023, DA14580 interfacing with external memory, Dialog Semiconductor

# 3 Introduction

This document provides an overview of the booting sequence of DA14580 and describes the steps for creating a secondary bootloader application. The application structure, the steps for testing it and a methodology for measuring the booting time are presented.

The secondary boot loader allows DA14580 to boot from a SPI or EEPROM flash memory or UART interface. It can be used to replace the ROM boot loader in case a faster booting sequence is needed.

An extension of the secondary bootloader based on dual images is also represented in this document. The dual image bootloader can be used together with the Software Upgrade Over The Air (SUOTA) application for the product firmware upgrade.

# 4 How DA14580 Boots

DA14580 operates in two modes [1, 2], namely the “Normal Mode” and the “Development/Calibration Mode” hereafter address as “DevMode”.

At power up or reset of the DA14580, the primary boot code (ROM code) will check if the OTP memory is programmed. When this is the case the DA14580 enters *“Normal Mode”*. It proceeds with mirroring the OTP contents to System RAM and it will program execution.

Otherwise, it enters *“DevMode”.* It scans a predefined number of pins to communicate with external devices, using the three interfaces available on chip:

- UART,
- SPI,
- I2C.

Due to the time it requires to scan the interfaces (SPI as slave, UART, SPI as master and I2C) in DevMode the start-up time is longer than in Normal Mode. The difference in start-up time is around a few milliseconds in Normal Mode to several decades of milliseconds in DevMode [2].For applications requiring to run from an empty OTP and at the same time have a short boot time a dedicated secondary loader can be developed, that will skip the long sequence of scanning interfaces. Such a dedicated *secondary loader* will start communicating directly on a selected interface to download software into SRAM. This secondary loader has to reside in OTP making it operating in Normal Mode instead of DevMode.

The flow chart of the booting is shown in **Error! Reference source not found.**

# 5 Secondary Boot loader application

In this section three use cases of the secondary bootloader are described: booting form UART interface, booting from EEPROM and booting from SPI flash.

## 5.1 Booting from SPI master

It is possible for a host controller to act as a SPI master. In this case the DA14580 is the SPI slave. Via the host controller the booting can take place.

No further details are mentioned now, about this way of booting.

## 5.2 Booting from UART

The secondary loader application reads the UART RX Pin status with GPIO_GetPinStatus() command and if it is high (1), it starts the booting procedure from the UART interface.

The protocol for booting from UART is the same [2], as that of ROM boot code and it is implemented in function FwDownload() (**uart_booter.c**).

It starts with the DA14580 UART TX pin transmitting 0x02 (Start TX, STX). The external device is expected to answer a 0x01 (Start of Header, SOH) byte followed by 2 more bytes (LEN_LSB, LEN_MSB) which defines the length of the code to be downloaded (first byte is the least significant, second byte is the most significant). The DA14580 answers with 0x06 (ACK) if 3 bytes have been received and SOH identified or with 0x15 (NACK) if anything went wrong or the size if greater that the expected one.

**Table 1: Transmission sequence**

| DA14580 | Data direction | External device |
|---|---|---|
| 0x02 (Start TX, STX) | → | |
| | ← | 0x01 (Start of Header, SOH) |
| | ← | LEN_LSB |
| | ← | LEN_MSB |
| 0x06 (ACK) or 0x15 (NACK) | → | |
| Copy the data to System Ram | ← | #bytes send:<br>LEN_MSB * 8 + LEN_LSB |
| Calculate & Send the CRC | → | Read/Check the CRC |
| Branch to 0x20000000 (System Ram) | ← | 0x06 (ACK) or 0x15 (NACK) |

At this point connection has been successfully established and the SW code will start being downloaded. The next N bytes are received and placed into the System RAM.

Following the completion of the required code bytes, the boot code will calculate the CRC and send it over the UART. The booting sequence ends with by reading the value 0x06 (ACK) at the URX line. CRC is calculated by XORing every successive byte with the previous value. Initial CRC value is 0x00.

During the final step of the boot code, the UART GPIO pins are initialized to the default state. SYS_CTRL_REG is programmed to Remap to System RAM and apply a SW reset or perform a branch to System RAM. It is depended of the base address of the application code, so the system starts executing the code.

Booting from UART method can be used by the user to download the flash programmer application in order to upgrade the application firmware stored in SPI flash. This option is provided by the SPI programmer tool of Smart Snippets.

## 5.3 Booting from SPI

The secondary boot loader application initializes the SPI interface and the SPI flash memory if no UART is detected the options SPI_FLASH_SUPPORTED and SUPPORT_AN_B_001 are defined in the file bootloader.h. Then, it checks if the header described in AN-B-001 [2] (Table 2) is present by checking the first two bytes of the header for the signature (0x70, 0x50) and if a valid header is detected it copies a number of bytes equal to Code Size to System Ram and starts the user application.

**Table 2: SPI Header**

| Byte # | Field |
|---|---|
| 0 | Signature(0x70) |
| 1 | Signature (0x50) |
| 2 | Reserved |

| | |
|---|---|
| 3 | Reserved |
| 4 | Reserved |
| 5 | Reserved |
| 6 | Code Size LS Byte |
| 7 | Code Size MS Byte |
| 8 | Code Data[] |

## 5.4 Booting from an I2C/EEPROM

The secondary boot loader application initials the I2C/EEPROM flash memory if no UART is detected and the option EEPROM_FLASH_SUPPORTED and SUPPORT_AN_B_001 are defined in the file bootloader.h. Then, it checks if the header described in AN-B-001 (Table 2) is present by checking the first two bytes of the header for the signature (0x70, 0x50) and if a valid header is detected it copies a number of bytes equal to Code Size to System Ram. Finally, it verifies the checksum of the code data and if it matches the CRC field of the header and starts the user application.

**Table 3: EEPROM Header**

| Byte # | Field |
|---|---|
| 0 | Signature(0x70) |
| 1 | Signature (0x50) |
| 2 | Code Size LS Byte |
| 3 | Code Size MS Byte |
| 4 | CRC |
| 5-31 | Empty Bytes |
| 32 | Code Data[] |

# 6 Dual Image Bootloader

This section describes an extension of the secondary boot loader to support a dual image bootloader scheme which is used in SUOTA application for the product firmware upgrade in the field.

## 6.1 Non-volatile memory map

The non-volatile memory map to meet the needs of the dual image bootloader scheme is represented in Figure 1.

The first part is the bootloader , in case it’s stored in the non-volatile memory. The bootloader can also be stored in the OTP memory for faster booting time. The header defined in the AN-B-001 must be programmed before the image of the bootloader.

The images with the corresponding headers are stored at offset #1 and offset #2 which are defined in the product header. The product header is suggested to be programmed at the last sector of the non-volatile memory.

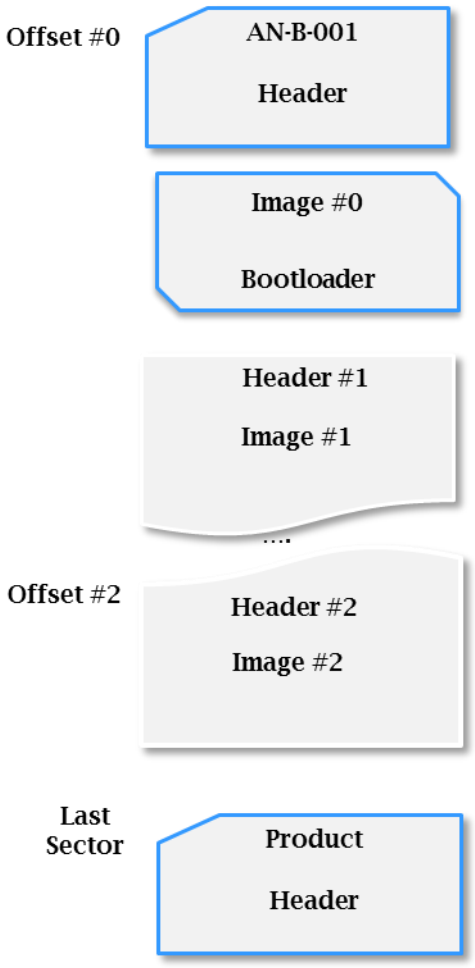


**Figure 1: Non-volatile memory map**

**Note 1** The bootloader is stored either in the first sector of the non-volatile memory according to AN-B-001 or in OTP flash memory.

The **product header** (Table 3) defines the addresses of the two firmware images stored in the non-volatile memory. It is programmed in the production line and the corresponding flash sector may be protected if it's supported by the flash characteristics.

- *signature (0x70, 0x52)*: It is a magic number identifying the product header .
- *version*: Two bytes reserved for the versioning of the product header
- *offset #1*: It defines the address of the first image stored in non-volatile memory. Four bytes in Little Endian format.
- *offset #2*: It defines the address of the second image stored in non-volatile memory. Four bytes in Little Endian format.

**Note 2** The product header may contain more information or configuartion settings i.e. NVDS data , DB address, XTAL16 trim settings. The use of these parameter is explained in the [6].

**Table 4: Product header format**

| Byte # | Field |
|---|---|
| 0 | signature(0x70) |
| 1 | signature (0x52) |
| 2-3 | Version |

| | |
|---|---|
| 4-7 | offset #1 in LE format |
| 8-11 | offset #2 in LE format |
| 12- | Reserved |

The **image header** includes the following fields as shown in Figure XX:

- *signature (0x70, 0x51)*: It is a magic number identifying the image header
- *validflag*: The validation field. It identify if the image has been downloaded correctly and the checksum has been verified. The value 0xFF defines a new image, the 0xAA defines a valid image and any other value defines an invalid image.
- *imageid*: An increment counter which defines the active image.
- *code_size*: It defines the size of the firmware image.
- *CRC*: It defines the checksum calculated over the image data.
- *version*: A 16 bytes string is used for the image version.
- *timestamp*: It defines the image creation time based on seconds since standard epoch of 1/1/1970.

**Table 5: Image header format**

| Byte # | Field |
|---|---|
| 0 | signature(0x70) |
| 1 | signature (0x51) |
| 2 | validflag |
| 3 | imageid |
| 4-7 | code_size in LE format |
| 8-11 | CRC in LE format |
| 12-27 | version (16 byte string) |
| 28-31 | timestamp in LE format |

## 6.2 Booting Sequence

During the booting phase, the bootloader checks the product header and reads the addresses of the two images stored in the non-volatile memory and reads the contents of the two image header to find out the valid image with the greater imageid and load it to system ram if the checksum of the code data matches the value of the CRC header field.

## 6.3 How to prepare the non-volatile memory

Smart Snippets tool can be used for burning the bootloader in non-volatile memory or OTP and the product header and the dual images in the non-volatile memory.

For example, the SPI memory preparation is done in 3 steps. The steps are described below:

1. Program the product header (0x1F000):

   Create a text file to describe the product header as shown below , load it using the Memory header option of the SmartSnippets toolkit, enter the following values in the product header fields

   Signature: 0x70, 0x52

   Version: 1234

   Offset1: 00800000 (it corresponds to offset 0x8000)

   Offset2: 00300100 (it corresponds to offset 0x13000)

and burn it to SPI flash at address 0x1F000. For more information how to create the product header refer to HELP menu of the SmartSnippets toolkit.

```
2       Signature     MagicNumber
2       Version       VersionNumber
4       Offset1       Offset_image_1
4       Offset2       Offset_image_2
```

2. Program the image binaries:

   Build the integrated processor proximity reporter application and convert the HEX to BIN file. Use the mkimage tool (located in tools\mkimage of the SDK distribution) to convert the BIN to IMG file. For more information how to use this tool, run mkimage without arguments.

   Load the image.img file using the SPI Flash Programmer option of the SmartSnippets toolkit and burn it to SPI flash at address 0x8000 (image #1) and 0x13000 ((image #2).

3. Program the bootloader:

   Build the secondary_bootloader as described below and burn the HEX file at address 0x0 using the SPI Flash Programmer option of the SmartSnippets toolkit. (Select 'Yes', in the question to make the SPI memory bootable.)

# 7 Application Description

## 7.1 System Initialization

The secondary loader application executes in the retention memory in order the System Ram memory to be available for storing the application data when the system boots either form UART or SPI interface. It sets the system clock and memory configuration as below:

```
SetWord16(CLK_AMBA_REG, 0x00); //fastest
SetBits32(GP_CONTROL_REG, EM_MAP, 7);
SetBits16(PMU_CTRL_REG, RETENTION_MODE, 0xF);
```

In the main function, secondary loader disables the Watch dog timer, sets all the peripherals in active mode and waits until the system is ready:

```
SetWord16(SET_FREEZE_REG,FRZ_WDOG);  // disable Watch Dog
SetBits16(PMU_CTRL_REG, PERIPH_SLEEP,0);   // exit peripheral power down
while (!(GetWord16(SYS_STAT_REG) & PER_IS_UP));  // Power up peripherals' power domain
```

For special designs when the system must work at 1.8Voltage, the following command must be enabled.

```
#ifdef SUPPORT_1_8_V
  SetBits16(DCDC_CTRL2_REG, DCDC_VBAT3V_LEV, 0x0);   ///--Support 1.8V boot
#endif
```

Next, the application detects the UART RX pin level and decides whether the UART or SPI booting function will be performed.

The file structure of the **secondary_bootloader** project is shown in Figure 2.

System initialization files: **startup_CMSDK_CM0.s, system_CMSDK.c, bootloader.sct** and **sysram.ini** are included in folder **secondary_bootloader/boot.**

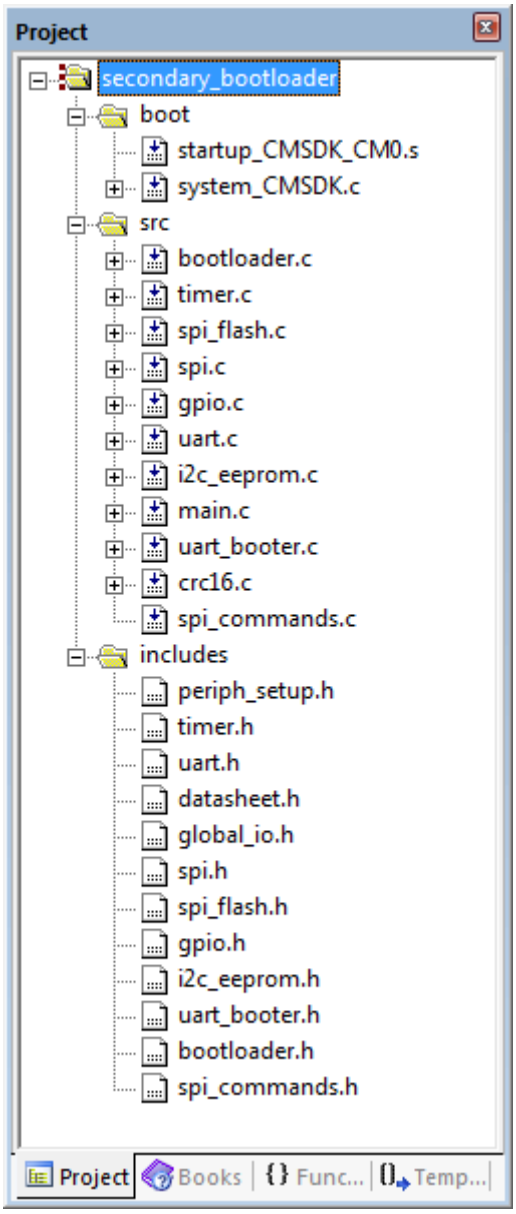


**Figure 2: File Structure**

- **startup_CMSDK_CM0.s:** The startup file for ARM Cortex-M0. It contains stack and heap configuration and the vector table.
- **system_CMSDK.c :** It contains the functions which initializes the system and update the SystemFrequency variable.
- **bootloader.sct :** It is the scatter-Loading description file
- **sysram.ini:** Keil debugger init script.
- Source files (.c): **main.c, bootloader.c, uart_booter.c, timer.c and uart.c spi_commands.c crc32.c** are included in the folder **secondary_bootloader/src.** Drivers files **spi.c, spi_flash.c and gpio.c** are located in folder dk_apps\src\plf\refip\src\driver\
- **main.c:** It contains the main function, system initialization function and the main loop of the application.
- **bootloader.c:** It contains the functions for booting from SPI and EEPROM and the implementation of the dual image bootloader.

- **Uart_booter.c:** It contains the functions for booting from UART.
- **timer.c.** In this file the timer functions are implemented. A software timer is used for the timeouts required by the UART boot protocol.
- **spi_commands.c**; It contains the additional commands for accessing the SPI flash.
- **spi.c, spi_flash.c, gpio.c:** Peripherals Drivers for SPI , SPI flash and GPIO interfaces. Detailed information for the drivers can be found in [3].
- Include files (.h): **periph_setup.h, timer.h and uart.h** are included in the folder **secondary_bootloader/includes**.
- **periph_setup.h:** This file contains the configuration settings for the peripherals (UART, SPI, SPI flash) used by the secondary boot loader application.

The main configuration settings are listed below:

```
// Select EEPROM characteristics
#define I2C_EEPROM_SIZE   0x20000         // EEPROM size in bytes
#define I2C_EEPROM_PAGE   256             // EEPROM's page size in bytes
#define I2C_SLAVE_ADDRESS 0x50            // Set slave device address
#define I2C_SPEED_MODE    I2C_FAST        // 1: standard mode (100 kbits/s), 2: fast
mode (400 kbits/s)
#define I2C_ADDRESS_MODE  I2C_7BIT_ADDR   // 0: 7-bit addressing, 1: 10-bit addressing
#define I2C_ADDRESS_SIZE  I2C_2BYTES_ADDR // 0: 8-bit memory address, 1: 16-bit memory
address, 3: 24-bit memory address

// SPI Flash settings
// SPI Flash Manufacturer and ID
#define W25X10CL_MANF_DEV_ID (0xEF10)     // W25X10CL Manufacturer and ID
#define W25X20CL_MANF_DEV_ID (0xEF11)     // W25X10CL Manufacturer and ID

// SPI Flash options
#define W25X10CL_SIZE 131072              // SPI Flash memory size in bytes
#define W25X20CL_SIZE 262144              // SPI Flash memory size in bytes
#define W25X10CL_PAGE 256                 // SPI Flash memory page size in bytes
#define W25X20CL_PAGE 256                 // SPI Flash memory page size in bytes

#define SPI_FLASH_DEFAULT_SIZE 131072     // SPI Flash memory size in bytes
#define SPI_FLASH_DEFAULT_PAGE 256        // SPI Flash memory page size in bytes

//SPI initialization parameters
#define SPI_WORD_MODE  SPI_8BIT_MODE
#define SPI_SMN_MODE   SPI_MASTER_MODE
#define SPI_POL_MODE   SPI_CLK_INIT_HIGH
#define SPI_PHA_MODE   SPI_PHASE_1
#define SPI_MINT_EN    SPI_NO_MINT
#define SPI_CLK_DIV    SPI_XTAL_DIV_2
// UART GPIOs assignment
#define UART_GPIO_PORT  GPIO_PORT_0
#define UART_TX_PIN     GPIO_PIN_4
#define UART_RX_PIN     GPIO_PIN_5
#define UART_BAUDRATE    baudrate_57K6

// SPI GPIO assignment
#define SPI_GPIO_PORT      GPIO_PORT_0
#define SPI_CS_PIN         GPIO_PIN_3
#define SPI_CLK_PIN        GPIO_PIN_0
#define SPI_DO_PIN         GPIO_PIN_6
#define SPI_DI_PIN         GPIO_PIN_5
// EEPROM GPIO assignment
#define I2C_GPIO_PORT   GPIO_PORT_0
```

```
#define I2C_SCL_PIN     GPIO_PIN_2
#define I2C_SDA_PIN     GPIO_PIN_3
```

W25X10CL SPI flash is supported. The W25X10CL arrays are organized into 512 programmable pages of 256-bytes each. Up to 256 bytes can be programmed  at a time. The W25X10CLhave 32 erasable sectors, 4 erasable 32KB blocks and 2 erasable 64KB blocks respectively. The W25X20CL SPI flash is also supported.

Other SPI Flash types can be supported by changing above configuration settings (i.e. SPI_FLASH_SIZE, SPI_FLASH_PAGE etc.)

GPIO Port 0 is used by default as it's supported by all DA14580 types (WLCSP, QFN40 and QFN48).

GPIO Pin 4 and 5 are assigned to UART TX and RX respectively.

GPIO Pin 0, 3, 5 and 6 are assigned to SPI CS, CLK, DI and DO respectively.

The conflict in GPIO Pin 5 is solved by sequential access to it from UART and SPI interface.

# 8 Getting Started

This section describes how to program the secondary bootloader and an application (i.e. integrated proximity reporter) in the SPI flash or the OTP memory and measure the system booting time. A comparison with normal booter (ROMbooter) is also provided.

Smart Snippets toolkit supports tools for SPI Flash and OTP memory programming. The SPI Flash must be programmed with proximity application and the OTP memory with boot loader application.

## 8.1 Writing application HEX file to SPI Flash

Smart Snippets SPI Flash Programmer is used for downloading an image file to the DA14580 SPI Flash Memory. Figure 2 shows the main screen of the SPI Flash programmer tool.

The following three steps are required for preparing the Proximity Reporter application:

Open Proximity Reporter project, *dk_apps\keil_projects\proximity\reporter_fh*.

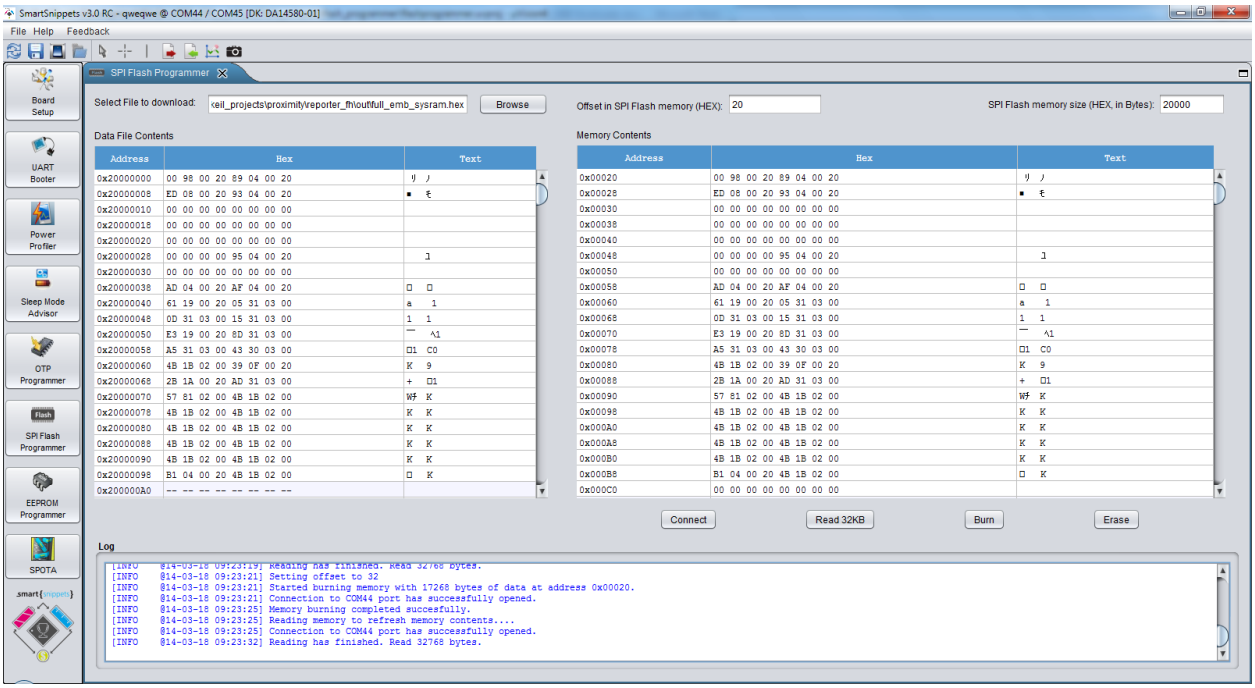


**Figure 3: SPI Flash Programmer**

Compile Proximity Reporter Application to generate the executable file full_emb_sysram.hex.

Open Smart Snippets and burn *full_emb_sysram.hex* to SPI Flash memory at **offset 0x20**. Header is automatically added by the Flash programmer firmware. SPI flash bootable for optimizing the boot time as the boot loader will copy only the actual application data written in SPI header. Make SPI flash non bootable to measure the max booting time as boot loader will copy 32KB from SPI flash.

## 8.2 Writing boot loader HEX file to OTP memory

OTP programmer tool enables downloading the default firmware into the System RAM and burning the OTP memory with a user-defined HEX and BIN file. Figure 3 shows the main screen of the OPT programmer.

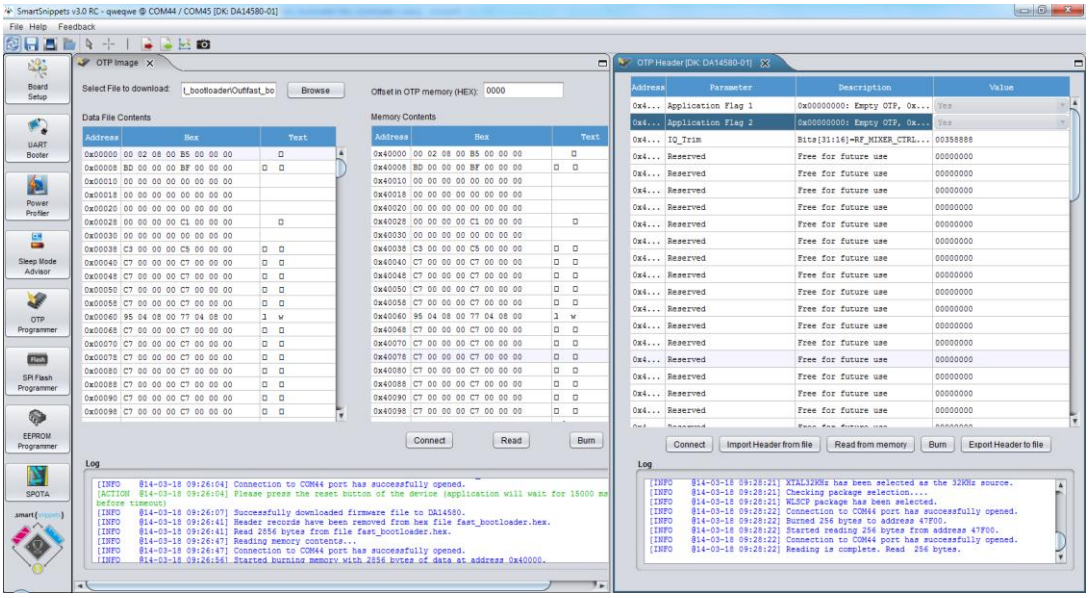

**Figure 4: OPT Programmmer**

Compile the project to generate the executable file secondary_bootloader.hex

The following steps are required for programming the executable secondary_bootloader.hex in OTP memory:

Open Smart Snippets application

burn secondary *bootloader.hex* in OTP image

Enable Application Flag 1 and Application Flag 2, set DMA Length and burn OTP header

**Note 3** J12 and J25 must be connected for OTP burning and UART interface, respectively. Make sure 6.8V is connected to the VPP pin. When doing this on the Dialog hardware development kit [3].

## 8.3 Measuring the booting time

This section describes the procedure for measuring the booting time of an application stored in SPI flash with the secondary boot loader described in this document and it's compared with the time required with the normal loader stored n ROM.

The Power Profiler tool of the Smart Snippets toolset is used to measure the time takes between power up and first advertising point.

In Figure 4, the booting time of the Proximity application with the secondary loader is shown. The point A corresponds to DA14580 power up time, point B corresponds to the first advertising point, points C and D illustrate the start and end point of the application data transfer from SPI to the system RAM and point C corresponds to the first application entry point (main()). The time required until the first advertisement is 158.8msec. The data transfer from the SPI flash takes 36msec (Note 2) and the time required until the first application entry point is 145msec.

**Note 4** 32KB data transferred instead of the actual application data size.

In Figure 5, the booting time of the Proximity application with the normal booting sequence is shown. The point A corresponds to DA14580 power up time, point B corresponds to the first advertising point, points C and D illustrate the start and end point of the application data transfer from SPI to the system RAM and point C corresponds to the first application entry point (main()). The time required until the first advertisement is 309.47msec. The data transfer from the SPI flash takes 100msec and the time required until the first application entry point is 295msec.

The secondary loader achieves faster boot time as it skips the scanning sequence of the normal mode while the SPI operation is optimized for the specific used SPI flash.

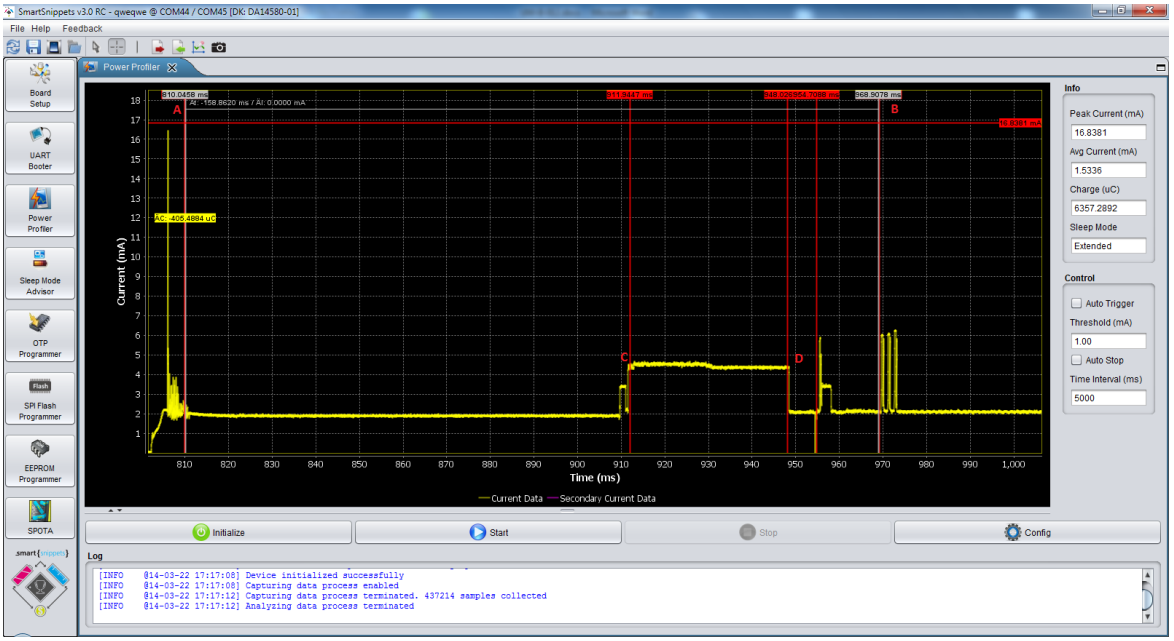


**Figure 5: Booting sequence using the secondary loader**

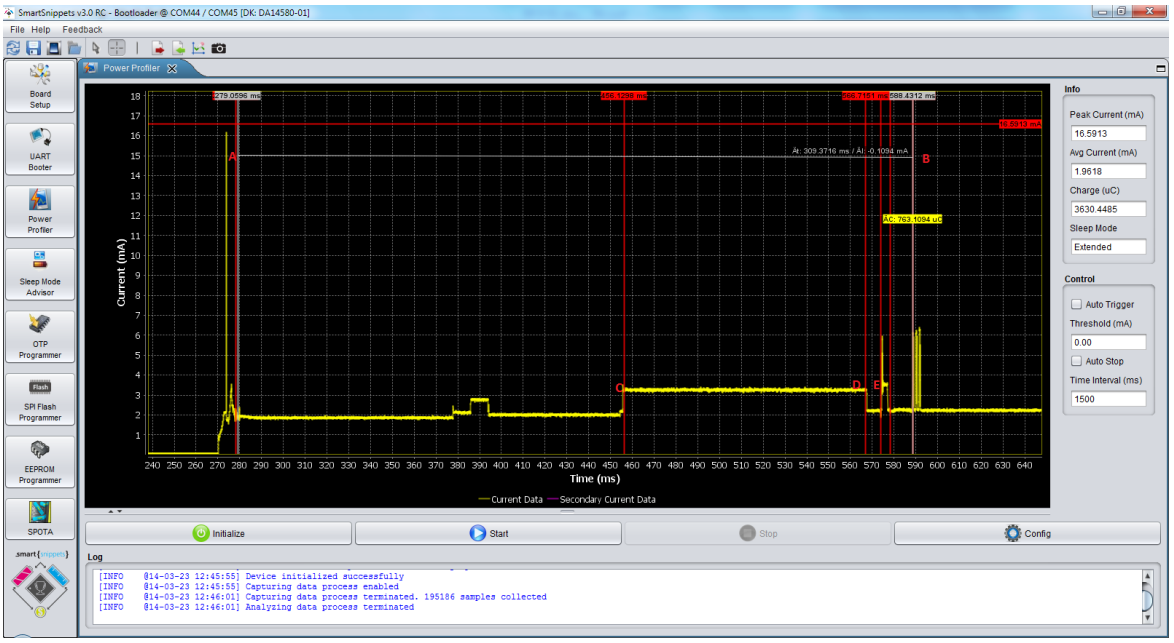


**Figure 6: Normal booting sequence**

# 9 Revision history

| Revision | Date | Description |
|---|---|---|
| 1.0 | 16-Jul-2014 | Initial version. |
| | | |

## Status definitions

| Status | Definition |
|---|---|
| DRAFT | The content of this document is under review and subject to formal approval, which may result in modifications or additions. |
| APPROVED or unmarked | The content of this document has been approved for publication. |

## Disclaimer



## RoHS Compliance

Dialog Semiconductor complies to European Directive 2001/95/EC and from 2 January 2013 onwards to European Directive 2011/65/EU concerning Restriction of Hazardous Substances (RoHS/RoHS2).
Dialog Semiconductor's statement on RoHS can be found on the customer portal https://support.diasemi.com/. RoHS certificates from our suppliers are available on request.

## Contacting Dialog Semiconductor

Germany Headquarters
*Dialog Semiconductor GmbH*
Phone: +49 7021 805-0

United Kingdom
*Dialog Semiconductor (UK) Ltd*
Phone: +44 1793 757700

The Netherlands
*Dialog Semiconductor B.V.*
Phone: +31 73 640 8822

Email:
enquiry@diasemi.com

North America
*Dialog Semiconductor Inc.*
Phone: +1 408 845 8500

Japan
*Dialog Semiconductor K. K.*
Phone: +81 3 5425 4567

Taiwan
*Dialog Semiconductor Taiwan*
Phone: +886 281 786 222

Web site:
www.dialog-semiconductor.com

Singapore
*Dialog Semiconductor Singapore*
Phone: +65 64 849929

China
*Dialog Semiconductor China*
Phone: +86 21 5178 2561

Korea
*Dialog Semiconductor Korea*
Phone: +82 2 3469 8291