RSL10 Bootloader Guide

1. OVERVIEW

The RSL10 bootloader provides means of performing firmware updates using the UART interface, and is a required component for Firmware Over the Air (FOTA). The bootloader enables firmware updates without the use of the JTAG interface. Firmware can be loaded from a host microcontroller over UART or over the air from another wireless device using FOTA. The bootloader copies the firmware image to the designated location in flash memory.

The bootloader source code is provided as a sample application in the RSL10 Software Development Kit (SDK), and comes pre-loaded on the RSL10 USB dongle. This dongle can be used for bootloader development but is not strictly required.

This document describes the bootloader firmware application and development tools. You will learn:

- How to customize the bootloader firmware
- How to create bootloader compatible applications and firmware images for RSL10
- How to use the UART updater PC tool to load a firmware image into the RSL10 USB dongle
- How the bootloader protocol between the bootloader firmware and the UART updater PC tool works with the RSL10 USB dongle

2. TERMS AND DEFINITIONS

CRC	Cyclic Redundancy Check
USB	Universal Serial Bus
CCIT	Causal Conditional Inference Tree
FOTA	Firmware Over The Air
UART	Universal Asynchronous Receiver-Transmitter
PC	Personal Computer
EVB	Evaluation Board

3. BOOTLOADER FIRMWARE

The bootloader firmware application must be initially loaded on to an RSL10 device. It can be loaded via JTAG. The RSL10 USB dongles come pre-loaded with the bootloader firmware.

Upon a power-on-reset, the bootloader determines whether to boot up the user application (default behavior) or activate the updater. The bootloader updater is activated on one of these three occasions:

1. A call to the `Sys_Boot_StartUpdater()` function from *sys_boot.h* is explicitly made from the user application.
2. The CFG_nUPDATE_DIO pin is held low while resetting the device (configurable in *config.h*).
   a. On the RSL10 Evaluation Board this can be accomplished by pressing both the reset and the user push buttons (DIO5) simultaneously, and then releasing the reset push button first. Upon releasing both push buttons, the LED (DIO6) on the RSL10 EVB is set constant high, indicating that the updater is active.
   b. On the RSL10 USB Dongle, this is performed by the *updater.py* PC tool. The red LED on the RSL10 USB Dongle stays on to indicate the updater is activated.

3. The bootloader detects an invalid user application in the flash memory.

In updater mode, the RSL10 UART is used to receive a *.bin* image file. If the image conforms to the expected format, the bootloader overwrites the user application area and reports a successful status. In case of error, no write operation is performed and an error status is returned over UART. Using a PC and the RSL10 EVB, the download of the *.bin* image file can be performed by the provided *updater.py* PC tool, as illustrated in Figure 1, below:

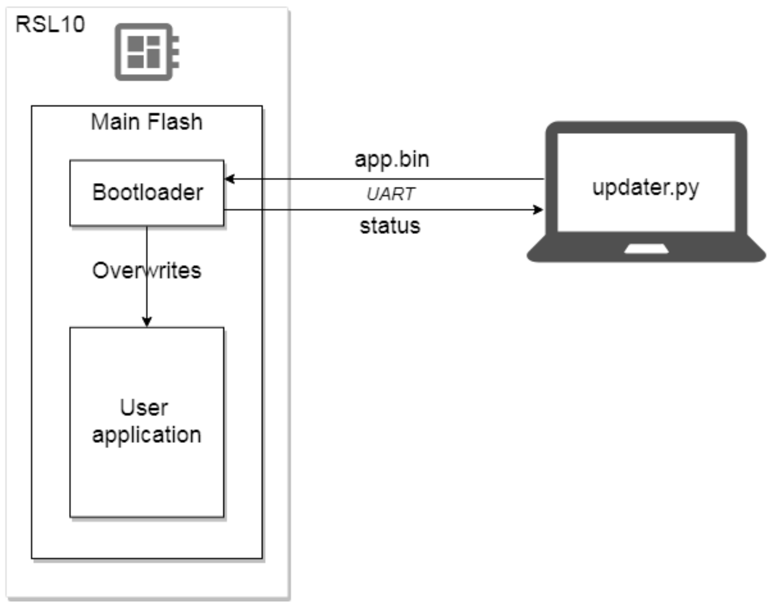


Figure 1. Updater Tool Downloads .bin Image File

CAUTION: The updater function of the bootloader runs from the PRAM. It is therefore theoretically possible to update the bootloader itself. However, we do not recommend this procedure, as your device can be irreversibly damaged if the power is lost during such an update.

The *config.h* header in the bootloader source code provides some configurable definitions, as shown below. It has pre-defined DIO and UART settings for both the RSL10 Dongle and the RSL10 EVB. You can choose between the EVB and the Dongle through the `RSL10_DEV_OR_DONGLE` definition.

```
#define RSL10_DEV_OR_DONGLE  RSL10_DEV    /* or RSL10_DONGLE */
#define CFG_nUPDATE_DIO      5            /* DIO pin number to activate the bootloader */
                                          /* during reset */
#define CFG_UART_BAUD_RATE   1000000      /* UART Baud Rate */
#define CFG_UART_RXD_DIO     4            /* UART RX pin */
#define CFG_UART_TXD_DIO     5            /* UART TX pin */
#define CFG_READ_SUPPORT     0            /* Enable/disable the support for read memory*/
                                          /* commands from the updater */
```

When the updater is activated, it expects to receive commands through the UART from the PC. If no command is received within a few seconds, the bootloader firmware watchdog times out and the device is rebooted.

3.1 Memory Map

The bootloader is placed at the base of the main flash region (address 0x00100000) and uses the first 8KB of memory. In order to be compatible with the bootloader, the linker memory map of a user application needs to be updated to add a new `BOOTLOADER` area and shift the application origin by 8KB (address 0x00102000). For example, the following memory map is the updated version of the one from the *blinky* sample application located in the *sections.ld* file:

```
MEMORY
{
    ROM (r) : ORIGIN = 0x00000000, LENGTH = 4K
    BOOT (xrw) : ORIGIN = 0x00100000, LENGTH = 8K
    FLASH (xrw) : ORIGIN = 0x00102000, LENGTH = 372K
    PRAM (xrw) : ORIGIN = 0x00200000, LENGTH = 32K
    DRAM (xrw) : ORIGIN = 0x20000000, LENGTH = 24K
    DRAM_DSP (xrw) : ORIGIN = 0x20006000, LENGTH = 48K
    DRAM_BB (xrw) : ORIGIN = 0x20012000, LENGTH = 16K
}
```

3.2 Application Version and ID

All firmware designed to be updatable by the bootloader must define a constant named `Sys_Boot_app_version` of type `Sys_Boot_app_version_t`. It is an 8-byte structure composed of a 6-ASCII-character-long application ID and a 16-bit version number:

```
typedef char Sys_Boot_app_id_t[6];
typedef struct
{
    Sys_Boot_app_id_t id;    /* App ID string */
    uint16_t num;            /* format: <major[15:12]>.<minor[11:8]>.<revision[7:0]> */
} Sys_Boot_app_version_t;
```

The macro `SYS_BOOT_VERSION(id, major, minor, rev)`, available in *sys_boot.h*, provides a convenient way to define this constant. This version constant is referenced in the Arm® Cortex®-M3 core interrupt vector table at position 7 (normally unused). As the interrupt vector table is always at the beginning of the firmware, it is possible to find the version information without knowing the symbol table of the particular firmware. This second listing is an excerpt from *startup.S*:

```
ISR_Vector_Table:
.long __stack            /* 0 Initial Stack Pointer */
.long Reset_Handler      /* 1 Reset Handler */
.long NMI_Handler        /* 2 Non-Maskable Interrupt Handler */
.long HardFault_Handler  /* 3 Hard Fault Handler */
.long MemManage_Handler  /* 4 Mem Manage Fault Handler */
.long BusFault_Handler   /* 5 Bus Fault Handler */
.long UsageFault_Handler /* 6 Usage Fault Handler */
.long Sys_Boot_app_version /* 7 Pointer to app version */
```

The user application can define any string ID. For the RSL10 USB Dongle, the IDs shown below in Table 1 are currently used as application IDs.

Table 1. RSL10 USB Dongle Application IDs

Application ID String	Meaning
BOOT_R	BootLdFW (Release configuration)
BOOT_D	BootLdFW (Debug configuration)
DONG_R	DongleFW (Release configuration)
DONG_D	DongleFW (Debug configuration)
HCI__D	HciFW (Debug configuration)

3.3 Image Format

The application firmware image to be updated via the bootloader must be created out of the *.elf* file by calling `objcopy`:

```
> arm-none-eabi-objcopy -O binary <app_name>.elf <app_name>.bin
```

For convenience, this can be added as a post-build command in your project settings. In Section 5.2, “Loading blinky.bin into RSL10 Using the PC Updater Tool (updater.py)” on page 6, we show how to add this project setting in Eclipse.

The following conditions must be fulfilled:

- The image has to be linked at the start of either the BOOT or the FLASH area.

CAUTION: If the image starts at the BOOT area, it replaces the bootloader, which can irreversibly damage the device.

- The image must fit into the flash memory.
- The image must be at least 1 KB in size.
- The image size must be a multiple of 8 bytes (this is automatically taken care of by the UART Updater PC application, *updater.py*, which pads the image).

4. UPDATER PC APPLICATION (UPDATER.PY)

The bootloader update protocol on the PC side is implemented by the Python script *updater.py*. This tool is available in the *scripts* subfolder of the *bootloader* sample application, and needs the following prerequisites:

- Installed Python, version >= 2.7 or >= 3.4
- Installed Python module *pyserial*, version >= 3.2
- *CP210xRuntime.dll* in the same directory as *updater.py*
- SiliconLabs VCP driver, version >= 6.7.3 (available from SiliconLabs website or in the RSL10 Bluetooth Low Energy Explorer installation, *ON Semiconductor/Bluetooth Low Energy Explorer/Driver/ CP210x_Windows_Drivers.zip*
- If you are using the RSL10 Evaluation Board, make sure you have the bootloader flash loaded and activated.

The following command displays the help message with instructions:

```
> python updater.py -h
```

usage: `updater.py [-h] [-v] [--force] PORT [FILE]`

This usage updates RSL10 with a firmware image file over UART.

Arguments that can be used with the *updater.py* command are shown in Table 2, below.

Table 2. Arguments for the updater.py Command

Argument	Type	Purpose
`PORT`	positional	COM port of the RSL10 UART
`FILE`	positional	image file (*.bin*) to download; without this parameter, the currently installed version info is printed
`-h, --help`	optional	show this help message and exit
`-v, --version`	optional	show program's version number and exit
`--force`	optional	force overwrite of the bootloader

The following command is an example of using the *updater.py* tool with the PORT positional argument to update the RSL10 Dongle firmware:

```
> python updater.py COM5 DongleFW.bin
```

This command gives the following output:

```
Image : DONG_R ver=1.0.1
Application: DONG_R ver=1.0.0
Bootloader : BOOT_R ver=1.0.0
****************************************************************************************
```

The command also programs the new firmware into the RSL10 USB Dongle. It shows the version information of the image file, the currently installed application, and the installed bootloader. For every transmitted flash sector of image data, an asterisk (*) is printed. More details about the bootloader protocol and the messages exchanged between the PC tool and the firmware are provided in Section 6., “The Bootloader Protocol” on page 8.

5. LOADING THE BLINKY SAMPLE APPLICATION USING THE BOOTLOADER

This section describes the process of preparing and loading the *blinky* sample application using the updater PC application (*updater.py*). This step-by-step tutorial assumes that you are using an RSL10 Evaluation and Development Board and have installed the pre-requisites described in Section 4., “Updater PC Application (updater.py)” on page 4.

5.1 Generating a Bootloader-Compatible Image of the Blinky Sample Application

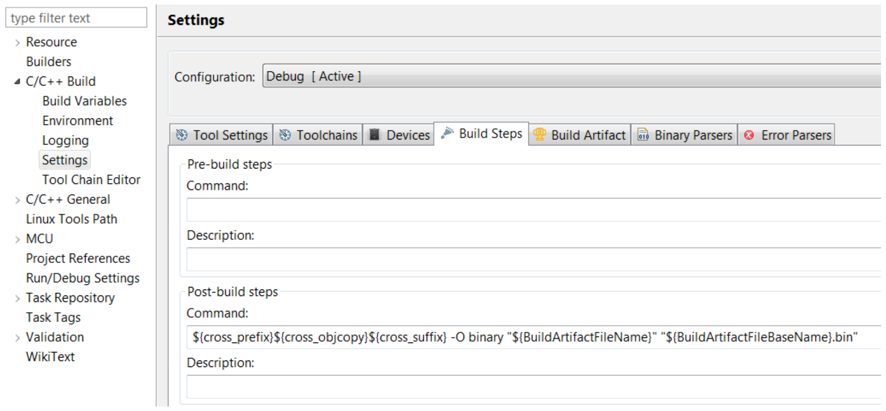
In order to generate a bootloader-compatible firmware image, you first need to modify the linker settings of the *blinky* application. Then, a post-build step is added to generate the bootloader binary image.

Step-by-step guide:

1. Import the *blinky* sample application
2. Modify the *sections.ld* linker configuration file to include the `BOOT` section and shift the start of the `FLASH` section by 8KB, as shown in Section 3.1, “Memory Map” on page 3.
    a. If you are using the CMSIS-Pack, this file is available in the *blinky* sample application folder.
    b. If you are using the standalone RSL10 SDK, this application uses the default *sections.ld* file, located in the configuration folder in your SDK installation. Copy this file into your *blinky* sample application folder before modifying it.

3. Make sure your project uses the modified linker script. Go to the project settings and navigate to **C/C++ Build** > **Settings** > **Cross ARM C Linker** > **General Script files (-T)**. It points to the local *sections.ld* file *${workspace_loc:/${ProjName}/sections.ld}*.
4. Add a post-build step to generate the binary (*.bin*) file required by the bootloader, as shown in Section 3.3, "Image Format" on page 4. In the project settings, navigate to **C/C++ Build** > **Settings** > **Build Steps** > **Post-build steps** > **Command** as shown below in Figure 2, and add the following command to generate the binary image:

```
${cross_prefix}${cross_objcopy}${cross_suffix} -O binary "${BuildArtifactFileName}"
   "${BuildArtifactFileBaseName}.bin"
```



Configuration: Debug [Active]

```
${cross_prefix}${cross_objcopy}${cross_suffix} -O binary "${BuildArtifactFileName}" "${BuildArtifactFileBaseName}.bin"
```

Figure 2. Adding a Post-Build Step

5. Set the ID and version number of your application by adding the following lines of code in the *app.c* file:

```
#include <sys_boot.h>
SYS_BOOT_VERSION("BLINKY",2,0,0);
```

6. Build the *blinky* application. If everything is correct, a *blinky.bin* file is generated in the output folder. This is the binary image file required by *bootloader*.

5.2 Loading blinky.bin into RSL10 Using the PC Updater Tool (updater.py)

1. Import and build the *bootloader* sample application.
2. Make sure your RSL10 EVB is connected to your PC, and use a debug session to flash load the *bootloader.elf* file into RSL10's flash memory.
3. Reboot your board. The RSL10 EVB LED (DIO6) stays on, indicating that the bootloader updater is activated, as there is no valid user application in flash yet.
4. Use the Windows™ Device Manager and locate the COM port assigned to your RSL10 EVB (COM40, in this example), as illustrated below in Figure 3.

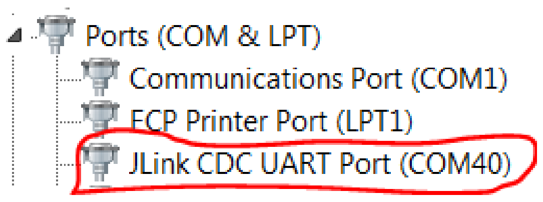


Figure 3. Locating the RSL10_EVB COM port

5. Copy the *blinky.bin* file generated in the previous section into the *bootloader/scripts* folder for ease of use.
6. Using the command prompt, navigate to the *bootloader/scripts* folder and use the *updater.py* tool to load the *blinky.bin* image into RSL10. You can expect an output similar to this one:

```
> python updater.py COM40 blinky.bin
Image      : ??????
Bootloader : BOOTL* ver=2.0.1
**
```

The "**" means that 2 flash sectors have been programmed and the image has been successfully loaded. The *bootloader* application reboots the board and the *blinky* application starts running (you can confirm it by observing the LED flashing).

If the output includes question marks after the word `Image`, as shown in the above example, it means that *updater.py* could not find the version information in the binary image. This could happen if the `ISR_Vector_Table` in *startup_rsl10.S* is not updated to include `Sys_Boot_app_version` in position 7, as shown in Section 3.2, “Application Version and ID” on page 3.

7. If the `ISR_Vector_Table` in *startup_rsl10.S* is correctly defined with `Sys_Boot_app_version` in position 7, and you run the previous command again, the output shows the version number of the installed application, the image, and the bootloader:

```
> python updater.py COM40 blinky.bin
Image      : BLINKY ver=2.0.0
Application: BLINKY ver=2.0.0
Bootloader : BOOTL* ver=2.0.1
***
```

If this error occurs,

```
> python updater.py COM40 blinky.bin
Image      : BLINKY ver=2.0.0
AssertionError: no data received
```

it is likely that *updater.py* could not communicate with the board because the *bootloader* updater has not been activated. As *bootloader* now finds a valid application in flash, it boots it up, instead of activating the updater. In order to force the activation of the *bootloader* updater mode, make sure to reset the RSL10 EVB by holding down the reset button and the DIO5 push button, and releasing the reset button first. After that, the LED on the board (DIO6) stays ON, indicating that the *bootloader* updater mode is active (it stays in this mode for a few seconds before timing out and rebooting the board if no serial commands are received).

6. THE BOOTLOADER PROTOCOL

The bootloader protocol on the PC side is implemented in the *updater.py* tool. After activating the updater mode, the PC side has to query the bootloader version, the currently installed application version, and the flash memory sector size, with the `HELLO` command. Then the image is transferred and programmed using the `PROG` command. Once the programming is complete, the device is set to application mode.

A command can consist of several messages, but every message from the PC side must be confirmed by the RSL10 bootloader firmware before the PC side can send the next message. Except for the standard `RESP` message, every message is appended with a CCITT-CRC.

6.1 RESP

The standard response is a two-byte message. In the first byte is the type encoded: `0x55` stands for `NEXT` and `0xAA` stands for `END`. The second byte for `type = NEXT` is always `0`; for `type = END`, the second byte contains an error code:

- `0` = `NO_ERROR`
- `1` = `BAD_MSG`
- `2` = `UNKNOWN_CMD`
- `3` = `INVALID_CMD$`
- `4` = `GENERAL_FLASH_FAILURE`
- `5` = `WRITE_FLASH_NOT_ENABLED`
- `6` = `BAD_FLASH_ADDRESS`
- `7` = `ERASE_FLASH_FAILED`
- `8` = `BAD_FLASH_LENGTH`
- `9` = `INACCESSIBLE_FLASH`
- `10` = `FLASH_COPIER_BUSY`
- `11` = `PROG_FLASH_FAILED`
- `12` = `VERIFY_FLASH_FAILED`
- `13` = `VERIFY_IMAGE_FAILED`

A message from the PC side with a bad CRC is always confirmed with `RESP(END, BAD_MSG)` by the device running the bootloader firmware. A command message with an unknown command code is confirmed with `RESP(END, UNKNOWN_CMD)`. A message with invalid parameters is confirmed with `RESP(END, INVALID_CMD)`. The standard response is the only message without an appended CCITT-CRC.

6.2 HELLO

The `HELLO` command message has no parameters, but because every command must be of the same length, the `HELLO` command message is padded with null bytes. The `HELLO` response message has three parameters:

1. The bootloader version **`<boot_ver>`** of type `Sys_Boot_app_version_t`
2. The version of the currently installed application **`<app_ver>`**, also of type `Sys_Boot_app_version` (if no application is installed, **`<app_ver>`** is filled with null bytes; if `app_version` in the interrupt vector table is 0, then the application ID of **`<app_ver>`** is set to `??????`).
3. The sector size of the RSL10 Flash memory in bytes

6.3 PROG

The `PROG` command message has three parameters:

1. Image start address

2. Image length in bytes
3. Image hash as Ethernet CRC32

If the start address and length are valid, the command is confirmed with `RESP(NEXT)`; otherwise, `RESP(END, INVALID_CMD)` is sent. After a positive confirmation, the PC side sends data messages containing the image data in sector-sized blocks of bytes, until it has sent the last data message containing the last part of the image. Every data message is confirmed with `RESP(NEXT)` until the last data message, which is confirmed with `RESP(END, NO_ERROR)`. If an error occurs during image transmission or programming, the next confirmation is a `RESP(END,` **`<error code>`**`)`. In this case, the PC side must start the whole sequence over again.

6.4 Restart

Similar to the `HELLO` command, the `RESTART` command message has no parameters, and the command message is padded with null bytes. If there is a valid bootloader, this command is confirmed with `RESP(END, NO_ERROR)` and the device is rebooted. Otherwise, it is confirmed with `RESP(END, NO_VALID_BOOTLOADER)` and no operation is performed. This command is usually executed after a successful firmware update, i.e., a sequence of `PROG` command messages.

6.5 Read

The `READ` command is disabled by default. The support for this command can be added by compiling the bootloader firmware with the configuration `CFG_READ_SUPPORT = 1` in *config.h*. This command has two parameters:

1. Start address to read from
2. Read length in octets (between 1 and flash sector size, 2KB)

This command is confirmed with `RESP(END, UNKNOWN_CMD)` if the debug lock is not set, or `RESP(END, INVALID_CMD)` if the length is invalid, i.e., length is zero or greater than `FLASH_SECTOR_SIZE`. Otherwise, the command returns the length in bytes read from the start address passed as argument, followed by a CRC.

Figure 4 on page 10 illustrates a typical message exchange between the bootloader firmware (running on RSL10) and the PC tool:

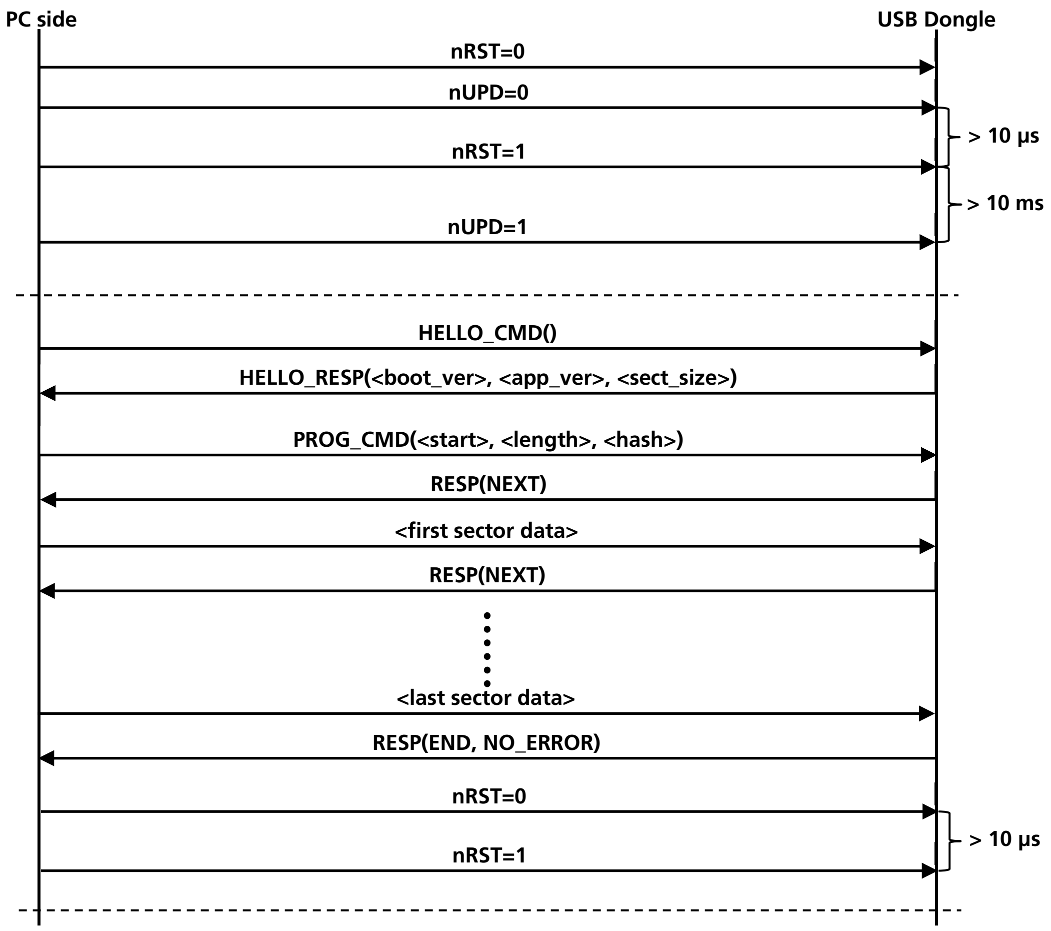


Figure 4. Sequence Diagram

PUBLICATION ORDERING INFORMATION

LITERATURE FULFILLMENT:
Literature Distribution Center for ON Semiconductor
19521 E. 32nd Pkwy, Aurora, Colorado 80011 USA
Phone: 303-675-2175 or 800-344-3860 Toll Free USA/Canada
Fax: 303-675-2176 or 800-344-3867 Toll Free USA/Canada
Email: orderlit@onsemi.com

N. American Technical Support: 800-282-9855 Toll Free USA/Canada
Europe, Middle East and Africa Technical Support:
Phone: 421 33 790 2910

ON Semiconductor Website: www.onsemi.com

Order Literature: http://www.onsemi.com/orderlit

For additional information, please contact your local Sales Representative