UCD3138 - Using Boot Flash and the Serial Port for Program Upgrades in the Field

Version 0.1
January 2014

Table of Contents

1 Introduction

This application note describes the bootflash firmware that can be used for field upgrading the power supply firmware in the UCD device over the isolation barrier. This application note provides a demo with detailed instructions on how to use the provided bootflash firmware.

The default size of the bootflash firmware is 2K Bytes. This size can be extended by making changes to the bootflash firmware implementation. For the UCD3138 the available code space for the user program (Power supply code) is 30K when bootflash is 2K. Because of the addition of the bootloader in the Flash, the user program has to be modified in order to make it functional and work together with the bootloader.

The bootloader firmware is complex; hence it is developed as a separate project in code composer studio.

1.1 Abbreviations and definitions

Term	Definition
User Program	The firmware used for implementing the power supply.
Pflash, program flash	Memory in flash used by the user program.
Bootloader	Bootloader firmware.
bootflash	Memory in flash used by Bootloader.
HyperTerminal	Serial interface program used in windows.

2 Evaluating the demo software

This section provides detailed instructions on using the demo firmware provided.

2.1 Prerequisites

Before using the demo firmware, the following items are needed:

1) Open-loop board/PFC EVM.
2) PMBus adapter.
3) Serial interface.
4) HyperTerminal or equivalent.
5) Device GUI.
6) .x0 files.
7) Test file for download.
8) Appropriate text files to send commands.

If using the HyperTerminal then set it up as shown in the below figure:

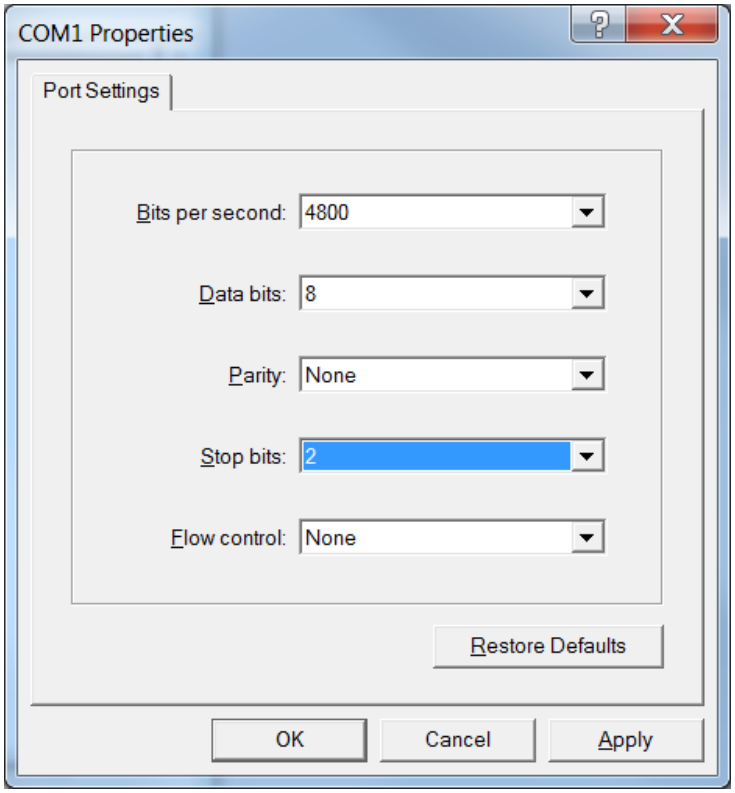

Figure 1: HyperTerminal settings.

For this demo, an Open-loop board (UCD3138OL64EVM-031) is used with the PMbus and RS-232 cable connected. On this board, UART 0 is used asTx and UART 1 is used as Rx(Connect J16 (2-3) and J17(1-2)). The entire setup used for this demo is shown below in the figure.

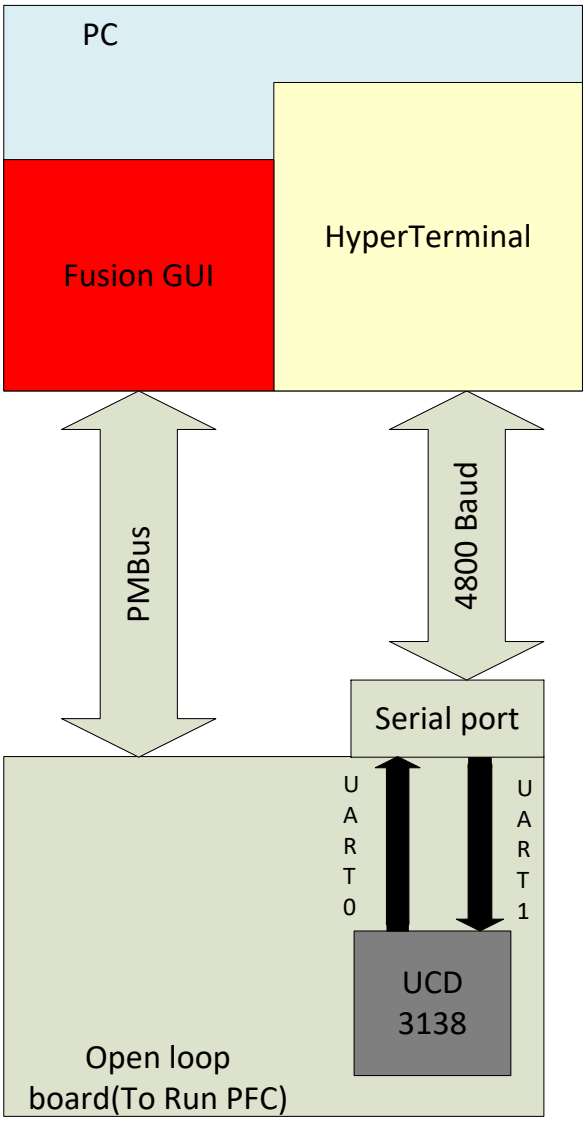

Figure 2: Setup for evaluating UART bootloader.

Open the device GUI and click on “Scan Device in ROM Mode”. Make sure that the device is detected in ROM mode. If not, then check if the SAA adapter is connected to the computer and then press the reset button on the board.

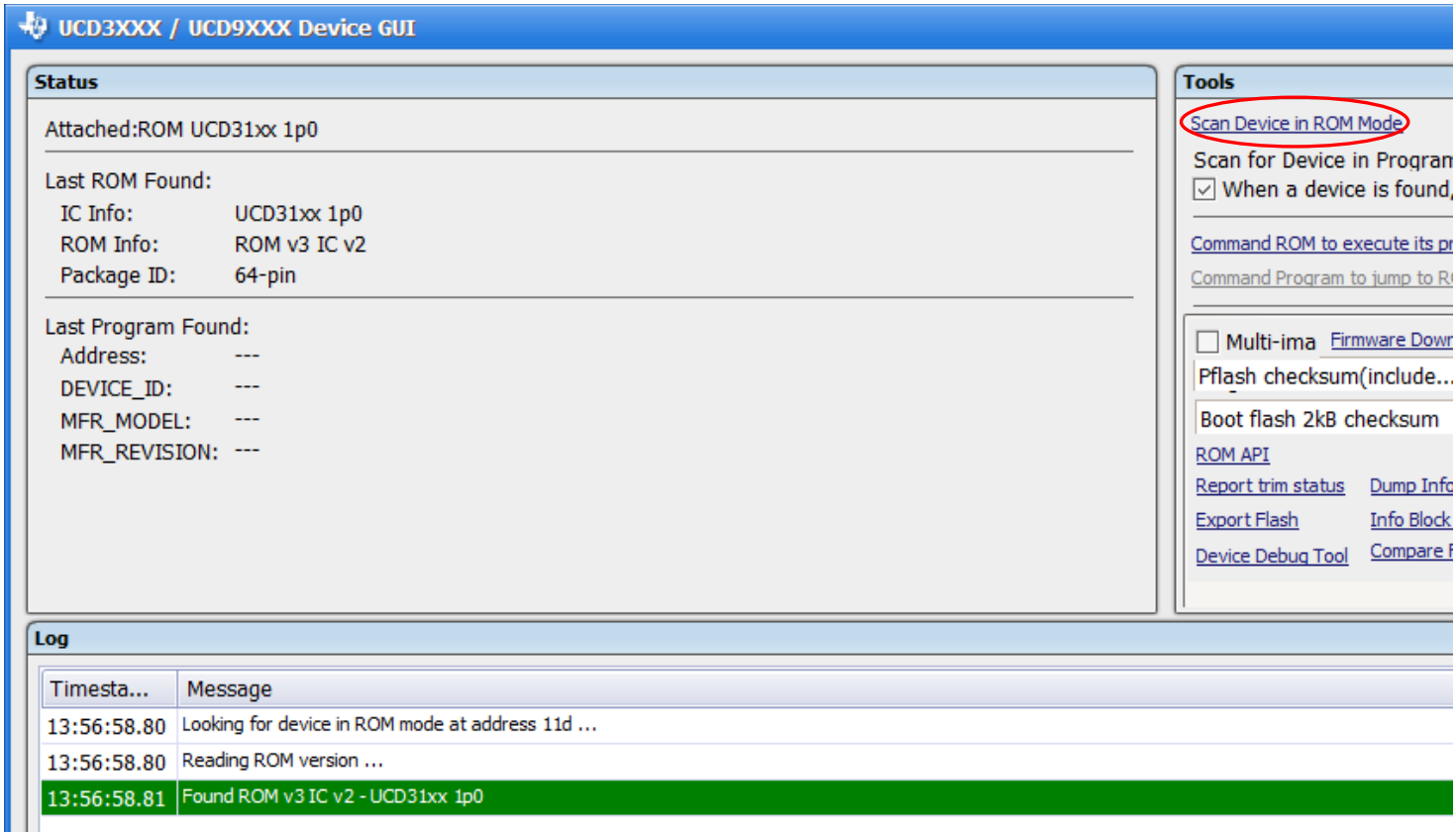

Figure 3: Detection of the UCD device using the Device GUI.

2.2 Downloading the bootflash code

Compile the bootflash project and download the .X0 file without writing the checksum as shown in the below figure.

Note: It is advised that the checksum is not written for demo purposes. In case the checksum was written and has to be cleared, please see section 2.7 to know how to clear the bootflash checksum.

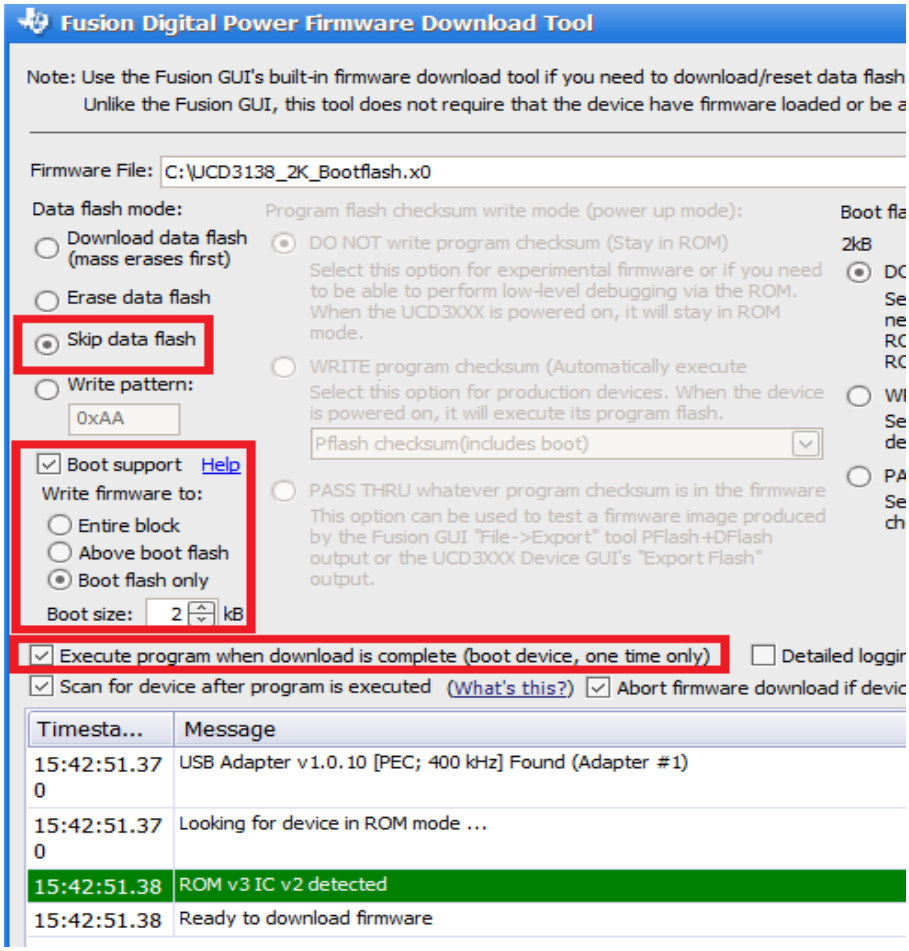

Figure 4: Download setting for bootflash

Once the download is complete, please check the HyperTerminal to see if it shows a message as shown:

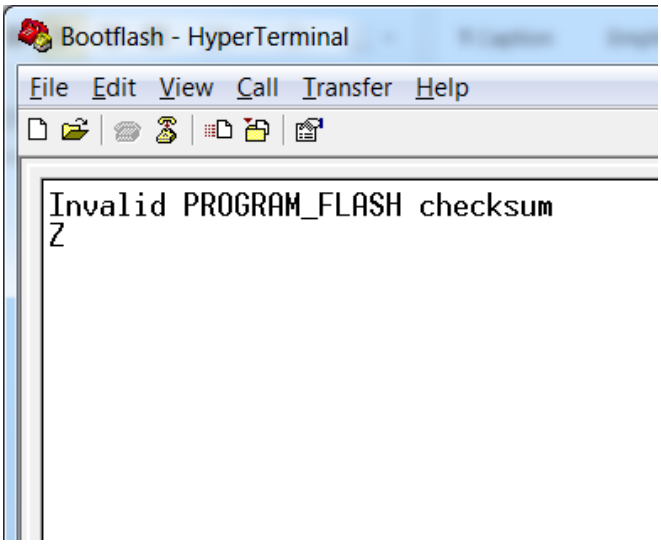

Figure 5: bootloader message after it starts running.

Now reset the device to allow downloading the user firmware (Power supply code. In this case a PFC firmware is used.)

2.3 Downloading the user code

Compile the user firmware project (PFC) and upload it through the device GUI without writing the checksum as shown:

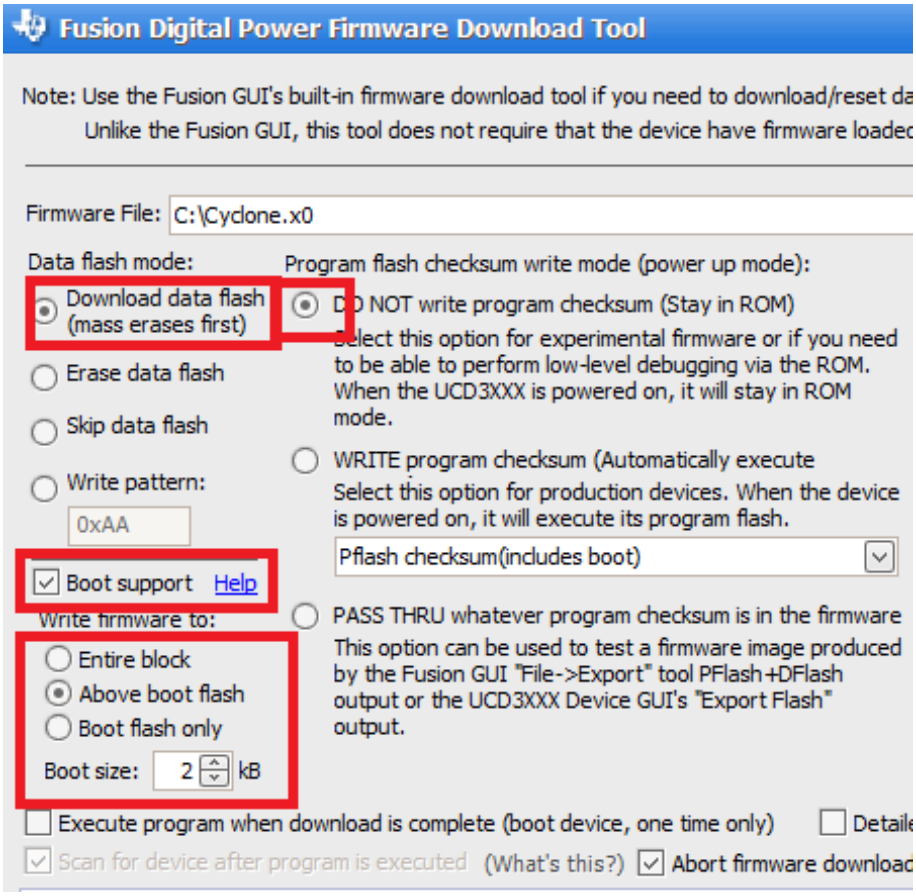

Figure 6 : Download settings for user program.

At this point the Flash contains both the bootloader program and the user program. But since there is no valid check sum for both the programs , none of the program will execute after reset.

2.4 Running the user code

Reset the device and click on “Command ROM to execute its program”. In the HyperTerminal window go to “Transfer -> Send Text File …” and select the “Execute_User.txt” file.

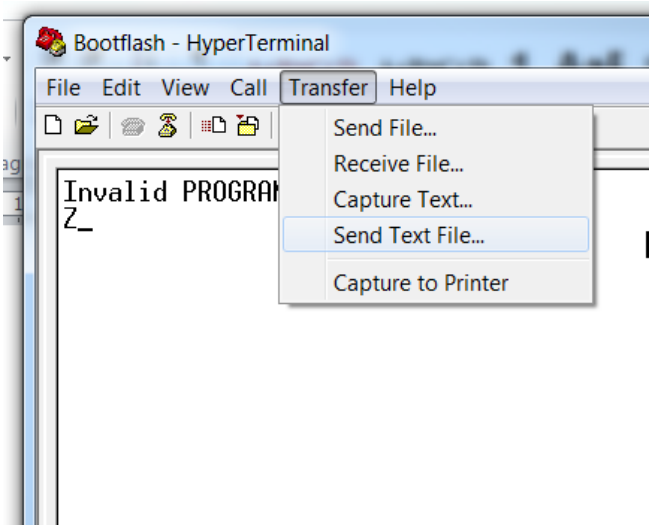

Figure 7: HyperTerminal option to send text files.

The “Execute_User.txt” file actually sends a command to the bootloader to jump to the user program and start running. When successful, a message as shown in the below figure should appear:

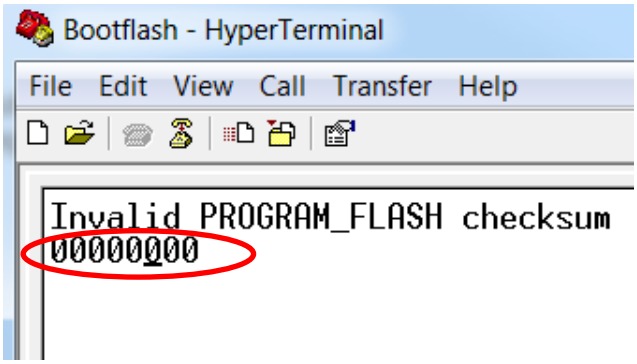

Figure 8: User Program execution .

To confirm the execution of the PFC code, click on the “DEVICE ID” in the device GUI and look at the status tab.

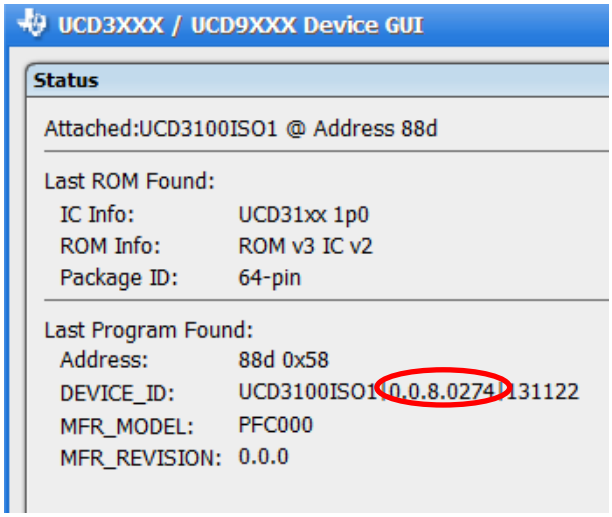

Figure 9: DEVICE ID of the user program.

2.5 Return to bootflash

To return back to the bootflash program from the user program, send the text file “Go_to_boot.txt”. If successful, a message “Z” should appear as shown in the figure.

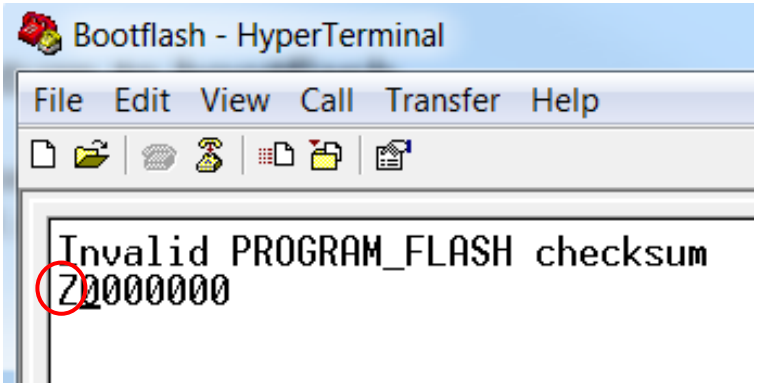

Figure 10: Message that indicates a return to the bootflash.

2.6 Uploading the user program using the UART

For field upgradability of the PFC code, the upload process has to be done using the UART. For this demo, the upload process is done using the HyperTerminal. Before the upload process can begin, the bootloader program has to be up and running. The demo code for the bootloader accepts only text files for uploading, hence the generated x0 file of the user project has to be converted to a text file that can be transferred through HyperTerminal. Included in the demo code is a converter program that is used to convert an x0 file to a text file that can be uploaded. The figures below illustrate how to proceed:

```
C:\temp>Post_Process.exe PFC_2K.x0 ht.txt 0 0
Do you want to place the checksum (y/n):n

INFO:
----
User program start address: 0x800
Program flash checksum    : 0x55fe2c
Checksum                  : Not Inserted
```

Figure 11: Using the converter program.

The converter program accepts the input as an x0 file and generates a text file that you specify. The options to this program are the byte space and the block space shown as '0' and '0' respectively in the above figure.

Open the *build.h* file in the user program project and increment the "BUILD_NUMBER". Now compile the project and convert the .x0 file as shown in the above figure. The output of the converter program is a text file called as ht.txt. Upload this file by using the send text file option in HyperTerminal. This will start the upload process and the HyperTerminal should display the following message when the upload process is finished.

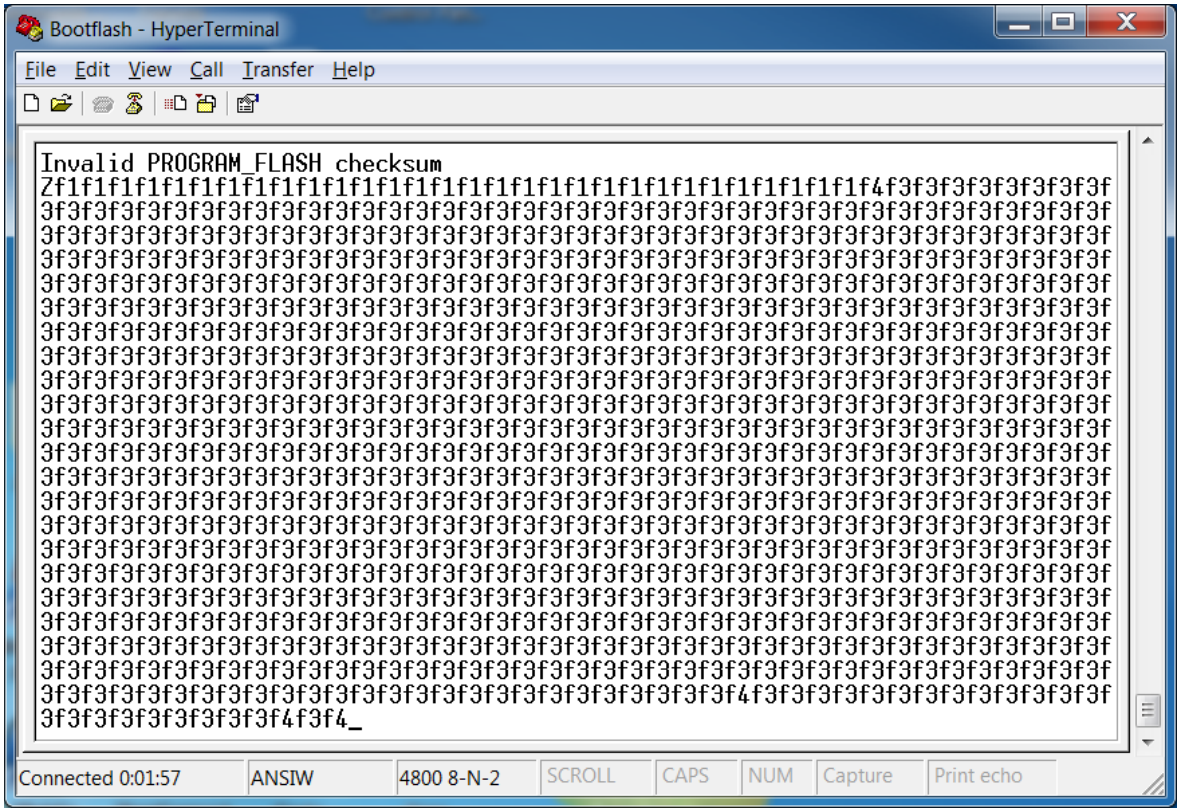

Figure 12: HyperTerminal output for upload process.

Once the upload process is over, clear the screen and send the “Execute_User.txt” file using the HyperTerminal. This will start the execution of the newly uploaded user program.

Click on the “DEVICE_ID” in the GUI to see the changes.

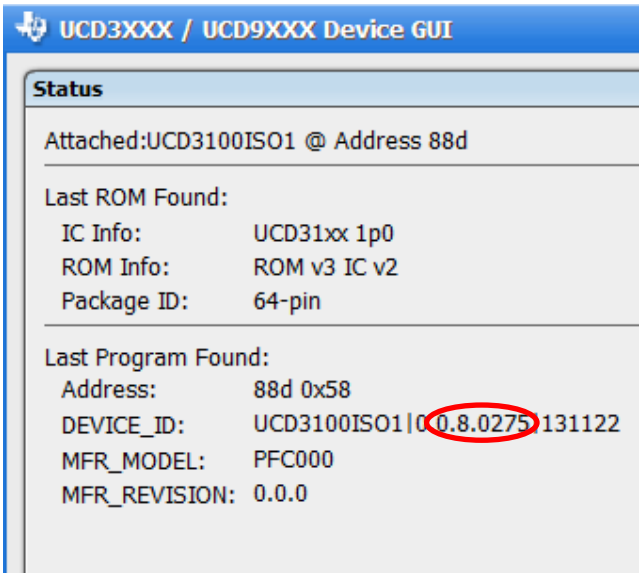

Figure 13: DEVICE_ID of the device.

2.7 Clearing checksums

The bootflash checksum can be cleared by writing a zero to the checksum location (0x07FFC). This can be done by using the “write word” command (0xF5) available in the bootloader. To clear the bootflash checksum send the file “Clear_bootflash_checksum.txt” over the HyperTerminal when the bootloader is running. Verify by resetting the device and check that the device does not jump to bootloader.

Similarly, use the file “Clear_userprog_checksum.txt” to clear the user program checksum. Please see **section 9** for more detailed description of this command.

3 Functions of Bootflash code

The bootloader is designed to perform tasks that help in updating and maintaining the user program. These can be summarized as mentioned below:

1) **Verify checksum of program flash memory**: Here the bootloader calculates the checksum by performing a bytewise addition of the Pflash and compares it with the value at 0x7FFC. If the calculated checksum matches with the stored checksum, the bootloader jumps to the user program; else, it displays a message on the terminal that the checksum is invalid.

2) **Erase main flash memory**: The bootloader before writing any data to the Pflash can perform a pagewise erase on the flash.

3) **Write to main flash memory**: The booatloader can write a word/ block of memory in the Pflash. Updating the user firmware involves a series of commands, data and checksum. This process can be started by the bootloader when the Pflash checksum is invalid or by the user program by jumping to the bootloader when an upgrade is needed.

4) **Execute user program**: The user program can be started by sending a specific command to the bootloader. This allows the user program to be started without writing the Pflash checksum.

5) **Clear checksums:** The bootloader does not have specific commands to clear checksums. However, with the help of already available commands like word write, the bootflash and Pflash checksums can be cleared.

6) **Read memory:** The bootloader can also read a word/ block form the memory.

7) **Calculate checksum:** The bootloader has a specific command that can calculate the Pflash checksum and display it on the terminal.

4 Changes to the existing user code to make it compatible with bootloader

4.1 Changes in load.asm:

The following lines have to be added to load.asm to make it work with the bootloader:

```
            .global rom_main
            .global _boot_upload_fw_vec
rom_main .equ 0xffffa4dc
```

```
; these three lines are added for ti_uart_boot_loader option
            .sect ".bvectors"
            ; boot vectors - mostly go to vectors in main program flash.
            .state32
_boot_upload_fw_vec
            .sect ".vectors"
            .state32
                  B     c_int00
```

4.2 Changes in cyclone.cmd

In the cyclone.cmd file of the UCD31xx family, there are two logical parts: a boot flash of 2K (0x0000 to 0x07fc), and a program flash of 30K (0x0800 to 0x7ffc). In order to make the command file compatible with the boot loader, make the following changes:

1) Remove the following line :

```
VECS            : org = 0x00000000, len = 0x00000020    /* Vector table
```

Add the following lines at the beginning of MEMORY:

```
BVECS           : org = 0x00000020, len = 0x00000004    /* Vector table */
PVECS           : org = 0x00000800, len = 0x00000020    /* Vector table */
```

Add the following line at the beginning of SECTIONS:

```
.bvectors       : {} > BVECS /* added for uart boot load*/
```

2) Make the following changes as shown :

i)
```
PFLASH     (RX) : org = 0x00000820, len = 0x00007738 /* PFlash Main Program         */
```

ii)
```
.vectors        : {} > PVECS
```

4.3 Change in other source files.

The only change required in the existing code is to provide a way to reenter back to the bootloader. The following changes shown are for an existing PFC code.

a) *Primary_secondary_communication.c* : Make the following changes in the function "**process_uart_rx_data"** as shown below:

```
if ((uart_rx_buf[0] & 0x80) == 0x80) //if program flash programming mode
            {
                  disable_fast_interrupt ();
                  disable_interrupt();

                  //turn off all PWM outputs
                  Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_VAL = 0;//drive low
                  Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_VAL = 0;//drive low
```

```
    Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_EN = 1;//turn off phase 1
    Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_EN = 1;//turn off phase 2.
    program_to_fw_update();// Added for bootflash support

}
```

a) *interrupts.*c : Add a new case as shown:

```
case 99:
    {
    boot_upload_fw_vec(); //vectored function call for flash
operations
    }
```

b) *software_interrupts_wrapper.c* : Add the following lines :

```
void program_to_fw_update(void)
{
    swi_single_entry(0,0,0,99);
}
```

c) *software_innterrupts.h* :Add the following line:

```
void program_to_fw_update(void);
```

Note: Please make sure that the Baud rate used by the user program matches with the bootloader. The bootloader by default uses a Baud rate of 4800.

5 Environment provided by ROM

The ROM in the UCD3138 device provides flexible ways to program the device. It can allow part of the flash memory to be used as a bootloader or the entire flash for a single program. The ROM provides this feature by looking at various checksum locations.

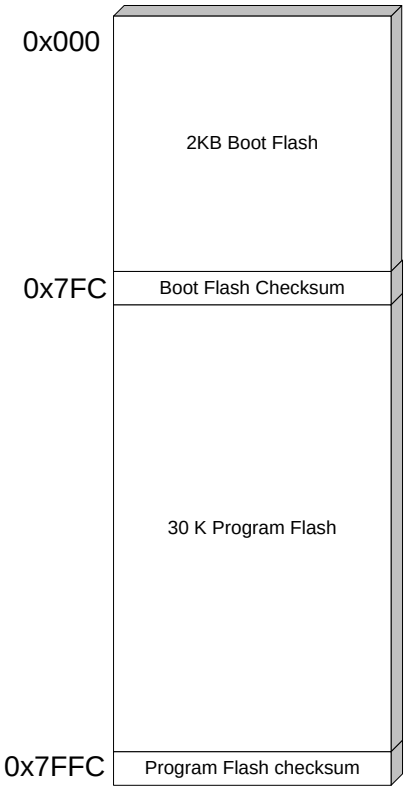

Figure 14: Checksum locations in UCD3138 for 2K bootflash.

A checksum is calculated by adding the bytes except the checksum bytes in a flash block. This checksum is then written in the last 4 bytes of the flash block. This is done when the flash is written for the first time.

After the UCD3138 powers up or reset, the boot ROM calculates 2 checksums. It first calculates a checksum for the first 2K bytes of program flash – from 0 to 0x7fb. If this checksum matches the 4 byte value found at 0x7fc, then the boot flash is enabled and in use. The boot ROM will move the program flash into location 0 and jump to location 0, starting execution of the boot flash program.

If the checksum at 0x7fc is not valid, the boot ROM will calculate the checksum for all 32K of the program flash - locations from 0 to 0x7ffb. If this checksum matches the 4 bytes at 0x7ffc, the boot ROM will also jump to program flash location 0. This checksum is used if boot flash is not required.

If both checksums are invalid, the boot ROM retains control and the program flash can be programmed via the PMBus.

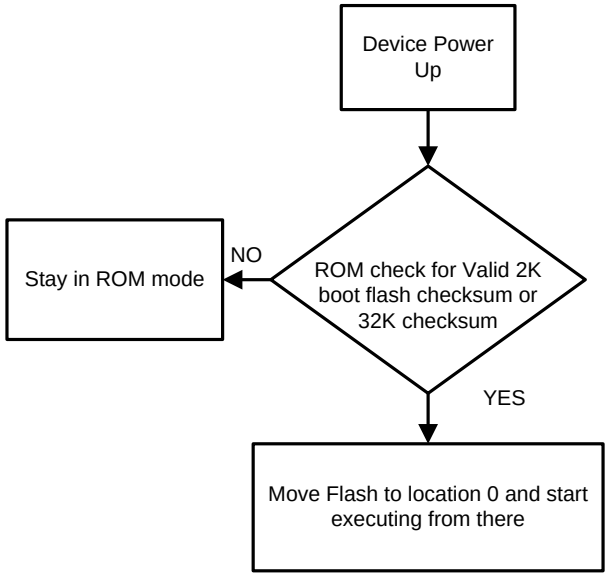

Figure 15: ROM functionality with bootloader feature.

6 Extended boot flash (boot flash size more than 2K)

The default Boot flash size is 2K. This size can be extended by making changes in the firmware implementation. In addition to the boot flash, all other flash sectors can be configured as part of the boot flash which can be extended up to 31 sectors (each 1K in size). To ensure firmware protection, the user firmware must create a custom checksum location at the end of the extended boot flash area. This will also require an update in load.asm and cyclone.cmd.

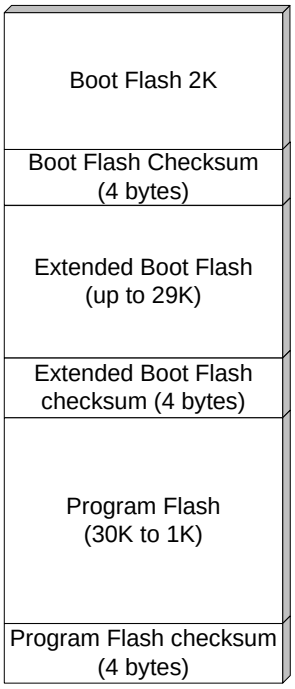

Figure 16: Flash structuring when extended bootflash is used.

Similar to the boot flash routine without the extended Boot flash size, at device reset the ROM will verify the 2K and 32K checksum and execute from boot flash location (location zero) if any valid checksum. The boot loader located in the first 2K size will check for the checksum located at the end of the extended boot flash. If the checksum is valid, the boot loader starts it normal operation. Otherwise, it clears the 2K checksum and resets the device.

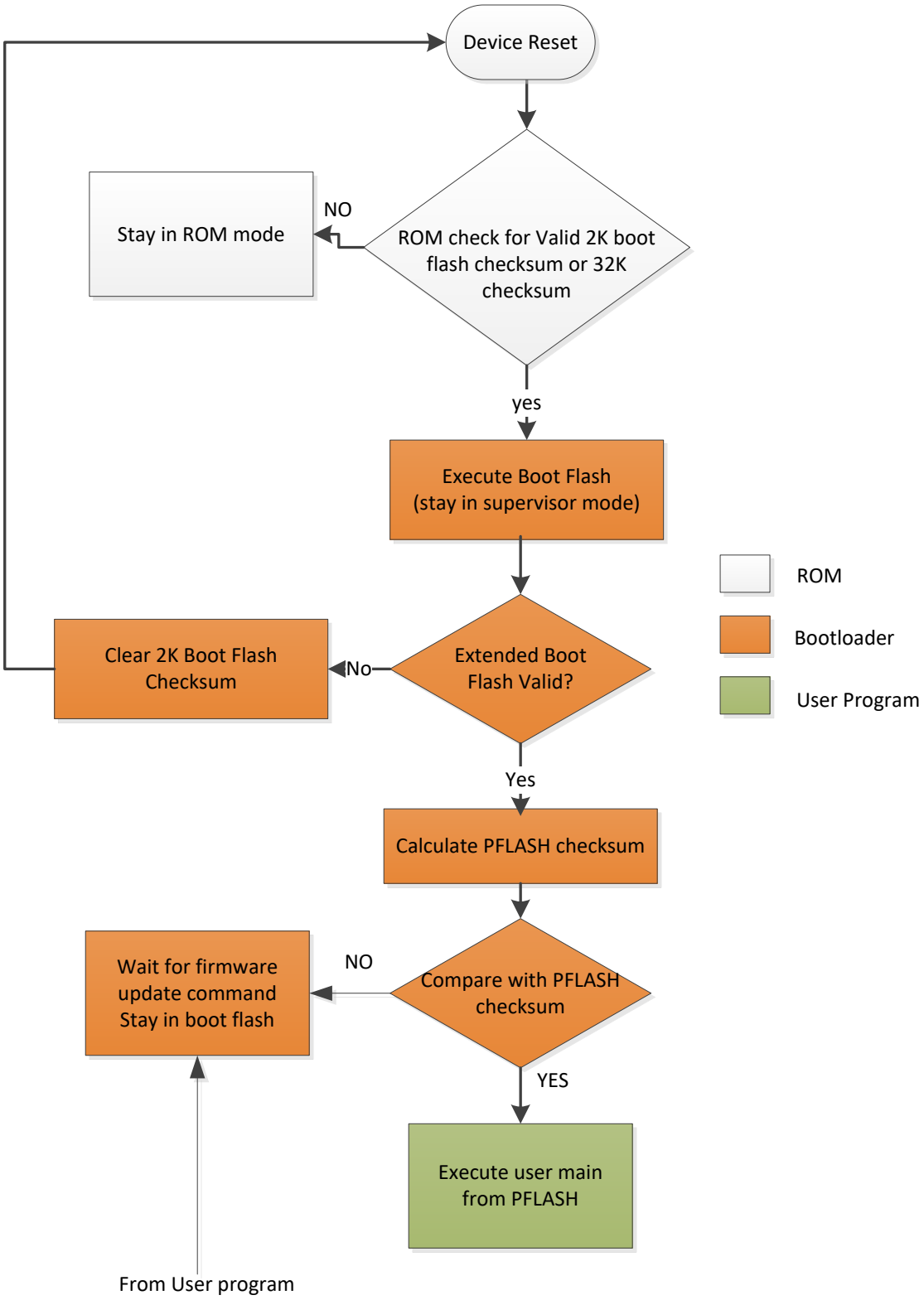

Figure 17: Flow chart for extended Bootflash.

6.1 Implementation of extended boot flash firmware

The user has the option to modify the boot flash size according to his/her requirements. In this example, the boot flash size is extended from 2K to 3K by making the following changes to the bootflash project:

- In **variables.h,** there is a preprocessor definition: **BOOT_FLASH_SIZE**. Change the value of this definition according to the boot flash size required.
- `#define BOOT_FLASH_SIZE (1024*3)`
- The **cyclone.cmd** linker file must be changed according to the change above.

When the user introduces a change in the boot flash size, the boot flash will be divided into two parts: The first part will be the original 2K boot flash and the second part will be the expanded bootflash. These two parts will be separated by a 2K checksum at the end of the first 2K boot flash. At the end of the expanded boot flash, there will be an additional checksum for the expanded boot flash.

Example: changes in "cyclone.cmd" for expanding boot flash to 3K size.

```
BVECS          : org = 0x00000000, len = 0x00000024    /* Vector table                */

/*--------------------------------------------------------------------------------------*/
/* ROM       8K    0xA000 - 0xBFFF                                                      */
/*--------------------------------------------------------------------------------------*/
ROM            : org = 0x0000A020, len = 0x00001D5E    /* System ROM                  */
SINE           : org = 0x0000BD7E, len = 0x00000282    /* Sine table                  */

/*--------------------------------------------------------------------------------------*/
/* B-Flash        2K   0x20  -  0x7FC
/* Ext. B-Flash   6K   0x800  - 0x1FFC                                                 */
/*--------------------------------------------------------------------------------------*/
BFLASH        (RX) : org = 0x00000024, len = 0x000007D8    /* Boot flash                        */
BFLASHCHK     (RX) : org = 0x000007FC, len = 0x00000004    /* Boot flash checksum               */
EXT_BFLASH    (RX) : org = 0x00000800, len = 0x000003FC         /* Extended Boot Flash          */
EXT_BFLASHCHK (RX) : org = 0x00000BFC, len = 0x00000004         /* Extended Boot Flash checksum*/

/*-------------------------------------------------------------------------------------- */
/* P-Flash   29K   0x0C00  - 0x7FFF                                                      */
/*--------------------------------------------------------------------------------------*/
PVECS1         : org = 0x00000C00, len = 0x00000004    /* Vector table                */
PVECS2         : org = 0x00000C04, len = 0x00000004    /* Vector table                */
PVECS3         : org = 0x00000C08, len = 0x00000004    /* Vector table                */
PVECS4         : org = 0x00000C0C, len = 0x00000004    /* Vector table                */
PVECS5         : org = 0x00000C10, len = 0x00000004    /* Vector table                */
PVECS6         : org = 0x00000C14, len = 0x00000004    /* Vector table                */
PVECS7         : org = 0x00000C18, len = 0x00000004    /* Vector table                */
PVECS8         : org = 0x00000C1C, len = 0x00000004    /* Vector table                */
```

After these changes, the user can put all extra boot flash code in the file called **extended_boot_flash.c** file. This file assembles in the expanded boot flash area using the linker file.

```
.texta              : {extended_boot_main} >         (EXT_BFLASH align(16))
```

7 Implementation of Boot Flash

The boot flash program is implemented as a separate project in Code Composer Studio. It is designed to occupy the first 2K of program space, and to use any RAM needed. After the other 30K is valid, control is turned over to that separate project. The main program needs to initialize its own stack pointers and RAM, just as if control was turned over from the boot ROM. The main program also has its own CCS project, as shown below:

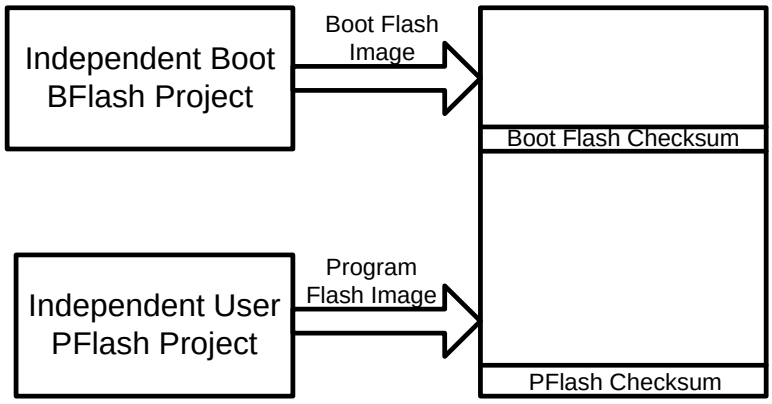

Figure 18: Independent Boot Flash and Program Flash Locations in Flash Memory

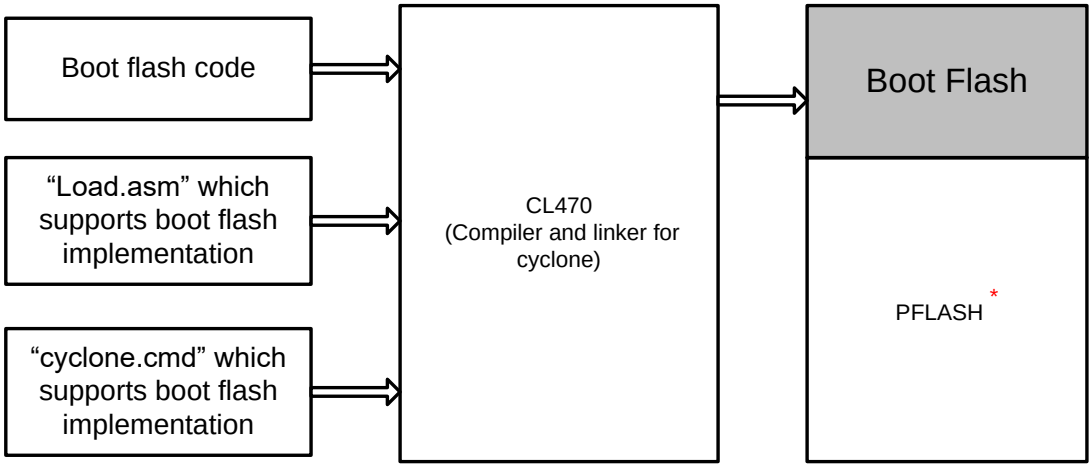

Figure 19: Compiling and downloading the boot loader program.

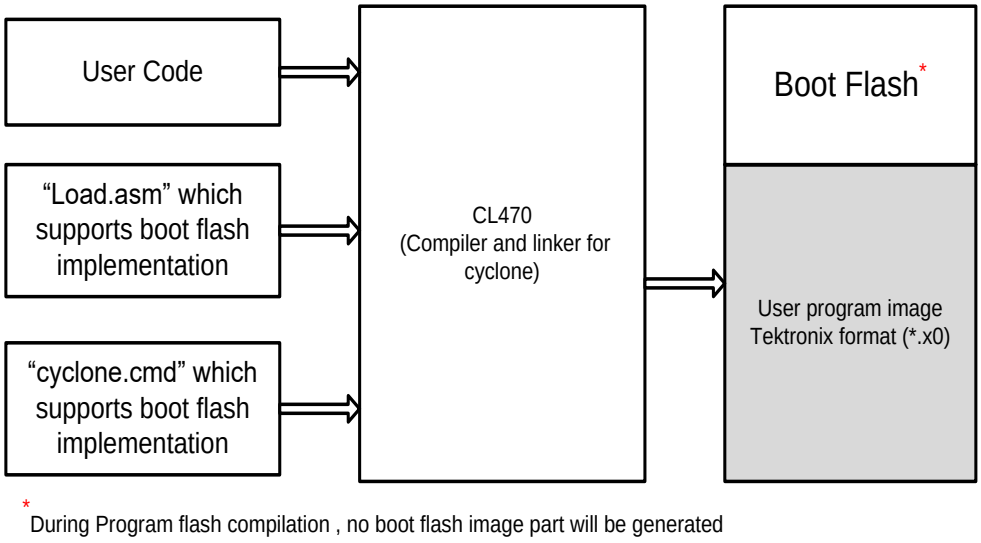

Figure 20: Compiling and downloading the user program.

Both the bootflash project and the user project uses a file called as "command" file. This file gives instructions to the linker to place specific functions/files in specific memory location. This file is carefully written to isolate the bootflash project from the user project.

Even though the projects are separated in memory, the processor is not aware of this separation. Hence the execution of code can be from either of the projects. However the execution of the bootflash code must be done in supervisor mode to perform flash manipulations. This is done by a software interrupt. Hence before transferring control to the bootlaoder a software interrupt is called from the user code.

8 Changes to existing user code to support extended bootflash

All the changes that were done in the PFC code for the 2K is applicable here except that the cyclone.cmd file has to be modified differently. These changes depend on the "**BOOT_FLASH_SIZE"** defined in variables .h. For an example value of "**BOOT_FLASH_SIZE"** to be 3K, here are the changes that should be made to cyclone.cmd file:

1) Make the following changes in the lines:

```
PVECS          : org = 0x0000C00, len = 0x00000020    /* Vector table            */

PFLASH    (RX) : org = 0x00000C20, len = 0x00007338    /* PFlash Main Program         */
```

2) Add the following lines :

```
EXT_BFLASH    (RX) : org = 0x00000800, len = 0x000003FC/* Extended Boot Flash       */
EXT_BFLASHCHK (RX) : org = 0x00000BFC, len = 0x00000004/* Extended Boot Flash checksum*/
```

9 Communication protocol used in UART

In order to place the user code (Instructions) at the correct address, a “command and checksum” protocol is implemented. It is simply a way of communicating to the boot loader and telling it to perform actions like place data, erase flash, get checksum and execute user program. The commands used are shown in the table below:

Message	Code	Number of bytes from secondary	Number of bytes from primary (not counting echo)
Send Read Address	0xFD	4	1
Read Next Block	0xF8	2	16
Write Word	0xF5	8	1
Write Block	0xF4	20	1
Write Next Block	0xF3	18	1
Mass Erase Flash	0xF2	2	1
Page Erase Flash	0xF1	3	1
Execute from Program Flash	0xF0	2	6
Read Checksum	0xEE	2	6

Table 2 (Command summary table)

The x0 file generated by the compiler contains only the address and the data. In order to place these commands, the x0 file has to be modified according to the protocol used. The converter program does this by parsing the x0 file and placing the commands and checksums appropriately.

The available converter program converts the x0 file to a text file by removing Tektronix specific address information and adding block writes with block address and checksums. It also places commands to perform flash erase before placing any block write commands. The converter places the commands in ASCII (or text) format; therefore each command takes up 2 bytes of data. The bootloader on the other side accepts all characters from ‘0’-‘9’, ’A’ - ‘F’ and ‘a’ - ‘f’ .It rejects all other characters like “Space”.

This format of UART communication was designed to make debugging easier. However, the time taken to upload new firmware through the UART would take twice as that would take by sending RAW (Hexadecimal) format.

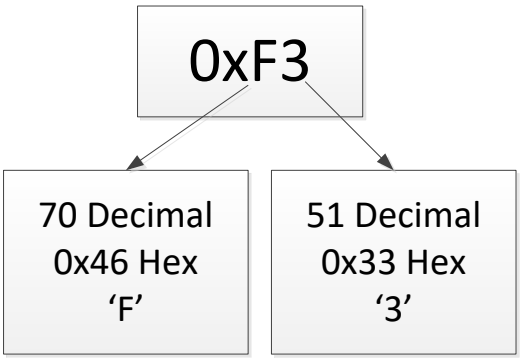

Figure 21: ASCII format for commands.

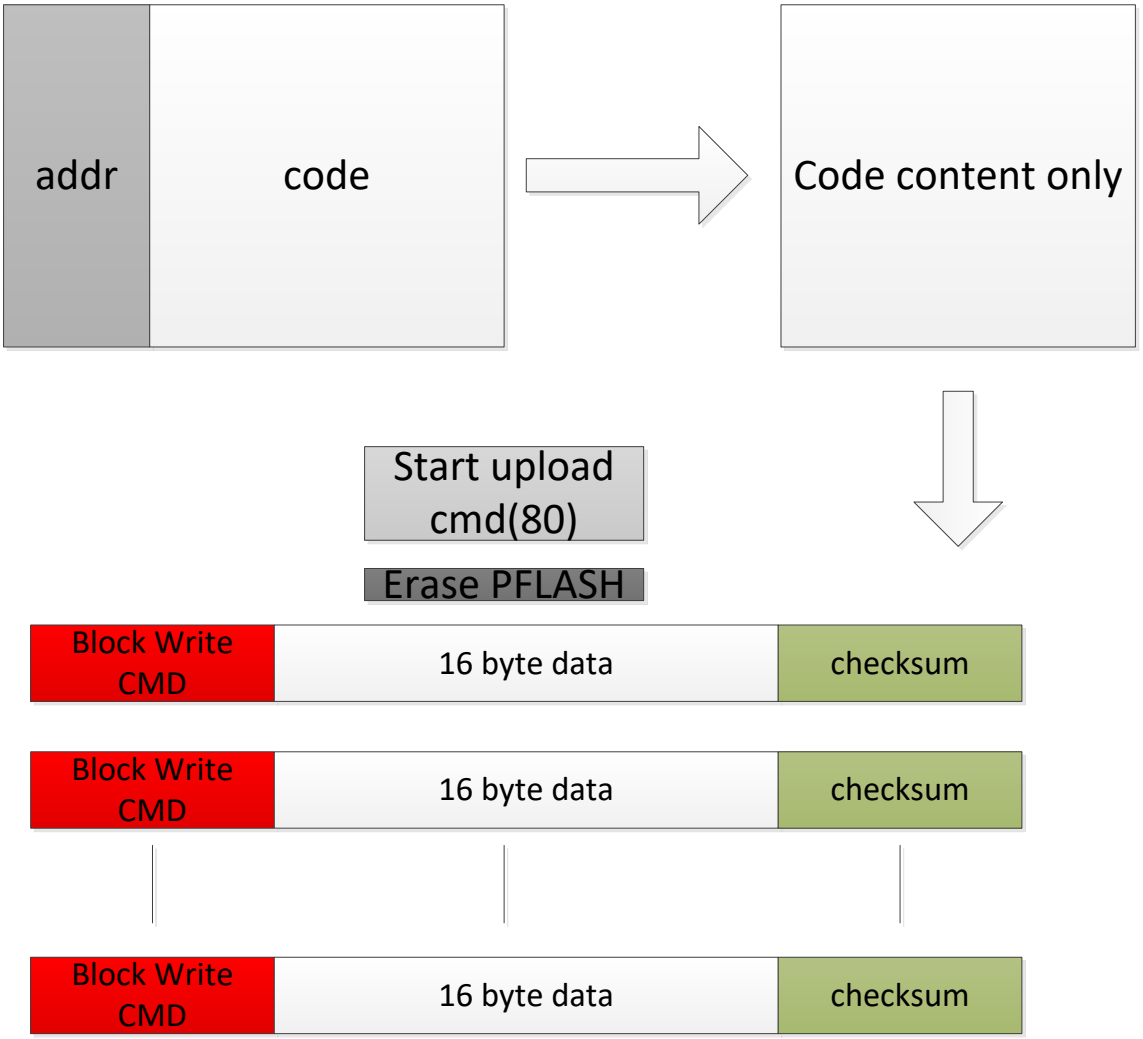

Figure 22: Conversion of x0 files to UART uploadable file.

9.1 Changing form text based UART upload to RAW data based UART upload

To speed up the upload process the user program can be uploaded in RAW (Hexadecimal) format. In order to make this happen the bootloader and the converter program has to be modified.

Make the following changes as mentioned:

1) All data received at **uart_text_buf** must be replaced with **uart_buf** in the bootflash program.
2) The converter program can be modified to add an extra feature that would convert the text based file to a desired hex format file.

10 Need of a bootloader in a PFC application.

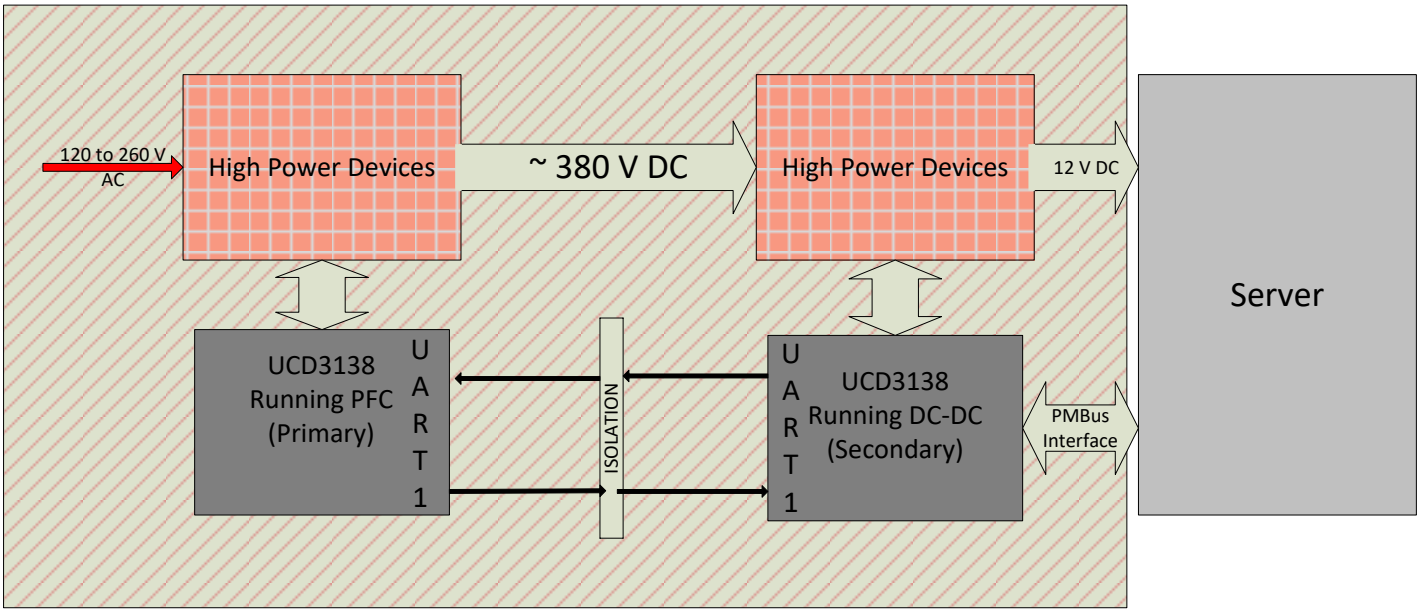

Figure 23: A Typical PFC application

The above diagram describes a typical PFC application. The AC line power comes into the PFC first. The PFC converts the AC to 380 volts DC. Between the input to the PFC and the output of the DC/DC, there is isolation of over 1000 volts. Finally, the DC/DC takes the high voltage from the PFC and steps it down to a lower voltage, often 12 volts, to be used by the server. This is a minimal AC/DC power supply.

Monitoring and communication are now common on AC/DC power supplies. The communication between the DC/DC and the host is carried on the PMBus interface, which is a standard power supply interface, derived from SMBus. But the PMBus is expensive to isolate, so a standard asynchronous serial bus (UART) is used to communicate between the DC/DC and the PFC.

Since the PMBus of the PFC is not accessible, firmware update is not possible through PMBus from the server. But since we have access to the UART of the PFC, firmware can be updated using a bootloader which uses UART for receiving a new firmware.

11 Combine two x0 files to a single one for production upload.

To speed up the process of upload for mass production, the two projects can be combined as a single x0 file and uploaded using the GUI. Here is how it should be done:

a) First, compile both the projects separately. Use the windows command "type file1 file 2 > combined_file" to concatenate both the files as shown below:

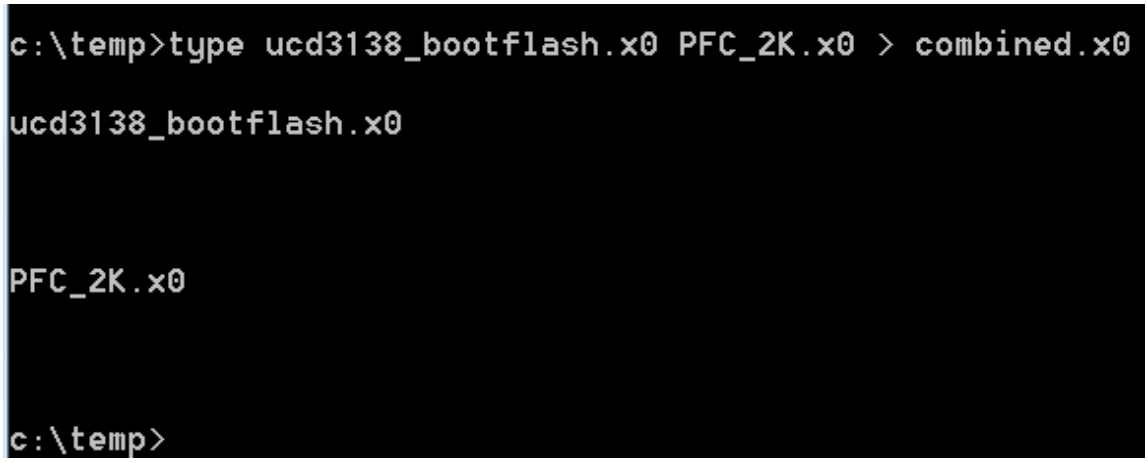

Figure 24: Combining two x0 files.

This new combined x0 file has code from both the x0 files. The above command format places the boot loader code first and then the user code.

b) Upload the combined x0 file using the fusion GUI and keep the options as shown below:

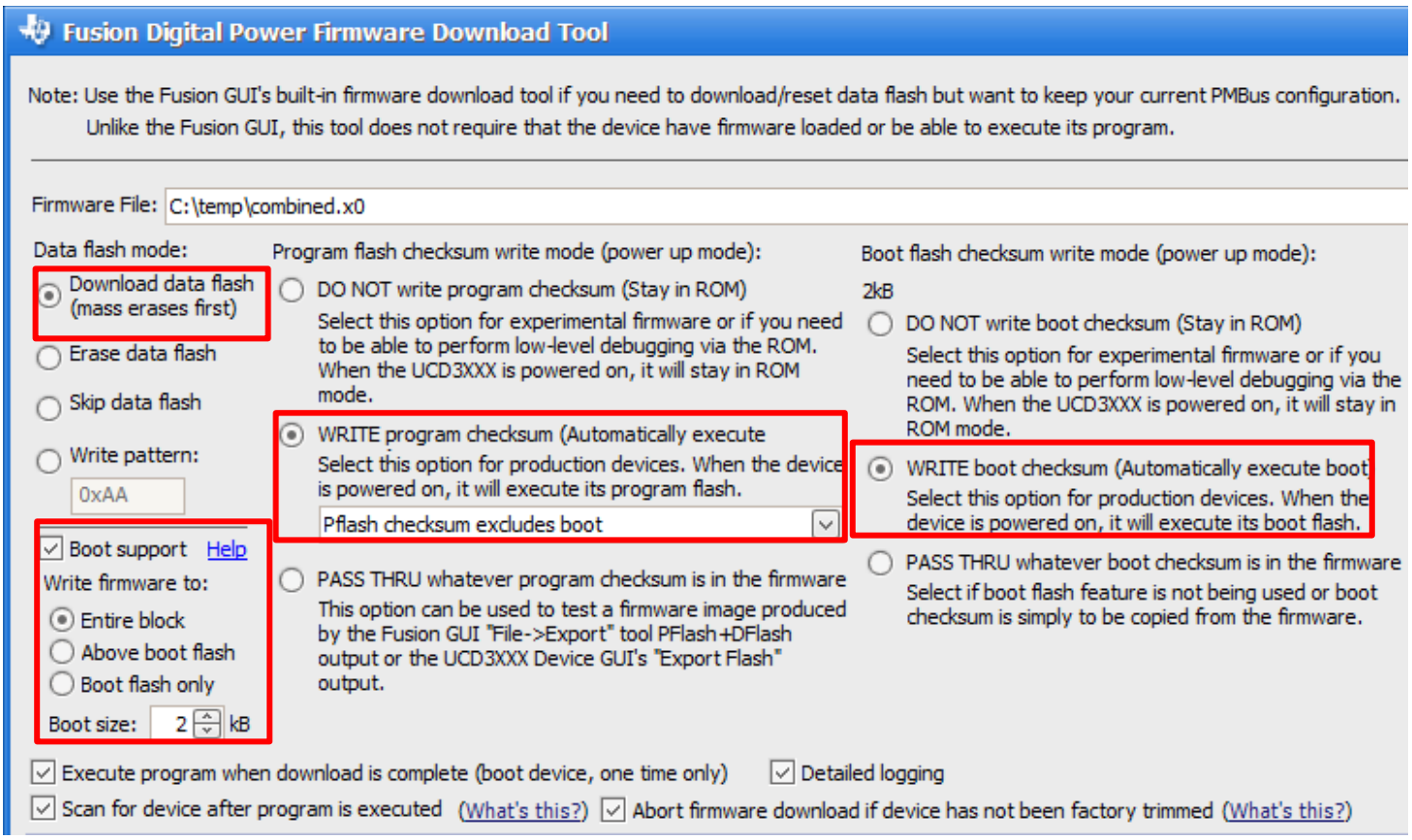

Figure 25: Download options for combined x0 file.