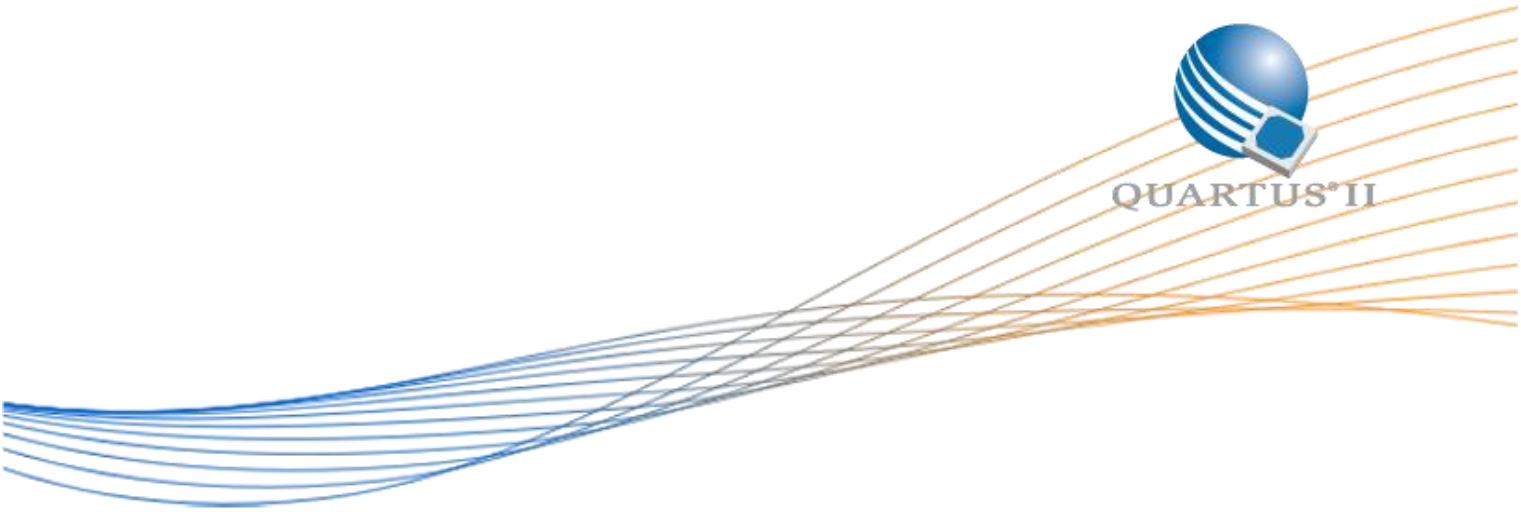


Utilizing the User Flash Memory (UFM) on Max 10 Devices with a Nios II Processor

Author: Devon Andrade

Date: 6/22/2015

Revision: 1.0

Theory of Operation

This design example utilizes a Nios II processor and the On-Chip Flash IP to modify and access the internal User Flash Memory (UFM). This guide will walk you through the basic concepts related to flash memory, the API used to access the flash, and the configuration modes available to the Max 10. The software included with this design example demonstrates how to read and write bytes to flash memory (without losing any data already within the flash) as well as how to erase blocks of flash memory. Lastly, there's a setup guide for both the hardware and software showing you how to compile and run the example.

Terminology

The following are important terms to know that make reading the Max 10 and Nios II handbooks easier to understand:

- **Erase:** set to '1'
- **Program:** set to '0'
- **Region/Sector:** both are used as terms to refer to a section of flash memory that contains multiple blocks/pages
- **Block/Page:** both are used to refer to a single erasable portion of flash memory

It's important to note that when reading "region" or "sector" inside of documentation that they refer to the same thing. This also applies to "blocks" and "pages." The Max 10 handbook tends to use the sector/page terminology while the Nios II software handbook (and documentation within the source code) likes to use the region/block terminology.

Software API

The Nios II Hardware Abstraction Library (HAL) supplies a host of functions that are used to access generic flash memory devices (this API could also be used to access external flash as well). Page 6-17 of the *Nios II Classic Software Developer's Handbook* describes the API used to access flash memory. In a nutshell, the main functions are shown below:

- `alt_flash_open_dev()` – Open a flash device
- `alt_flash_close_dev()` – Close an already opened flash device
- `alt_get_flash_info()` – Dynamically determine at runtime how many flash regions there are and the sizes of each region.
- `alt_erase_flash_block()` – Erase an entire page/block of memory.
- `alt_write_flash_block()` – write data to a page/block of memory without erasing it first.
- `alt_write_flash()` – Erase any affected pages/blocks then write to memory (may clobber data that was previously in the page/block).
- `alt_read_flash()` – Reads a variable length of bytes from flash memory into a buffer.

One thing to note is the difference between `alt_write_flash_block()` and `alt_write_flash()`. This being that `alt_write_flash()` will erase any pages that are about to be written to before writing, while `alt_write_flash_block()` does not. This has the potential of destroying any data that was in the same

block of memory, even if you aren't writing over it explicitly. To ensure that data isn't destroyed when using the `alt_write_flash()` function, it's recommended to perform a read-modify-write operation on the flash memory. Meaning, you read the entire page of memory into a buffer in RAM, modify the buffer with the data you want to include, erase the entire page, and write the buffer back to flash. This will ensure that no data is removed except the data you're explicitly overwriting. This process is demonstrated in the software available with this example inside of the `WriteByte()` function. An alternative to the read-modify-write process would be to store mutually exclusive pieces of information within their own pages (eliminating the need to read back the data before writing to flash) although this will only work if you have small amounts of data that need to be stored within the flash.

Max 10 Configuration Modes

How much User Flash Memory (UFM) that is available to your application is dependent on both the chip that was selected and the internal flash configuration that was selected. For every Max 10 device, there are five sectors/regions that are available for either user or configuration flash: UFM1, UFM0, CFM2, CFM1, CFM0. Despite having names that would suggest that certain sectors are only usable for user flash or configuration flash explicitly, which sectors are available as user flash is dependent on the selected configuration (shown further below).

Within each sector is a series of pages that can each be erased without affecting the other pages. Both the number of pages and how large each page will be is dependent on the selected chip. The table below (taken from the *Max 10 User Flash Memory User Guide*) describes the differences between chips:

Device	Pages per Sector					Page Size (Kb)	Total User Flash Memory Size (Kb) (1)	Total Configuration Memory Size (Kb)
	UFM1	UFM0	CFM2	CFM1	CFM0			
10M02	3	3	0	0	34	16	96	544
10M04	0	8	41	29	70	16	1248	2240
10M08	8	8	41	29	70	16	1376	2240
10M16	4	4	38	28	66	32	2368	4224
10M25	4	4	52	40	92	32	3200	5888
10M40	4	4	48	36	84	64	5888	10752
10M50	4	4	48	36	84	64	5888	10752

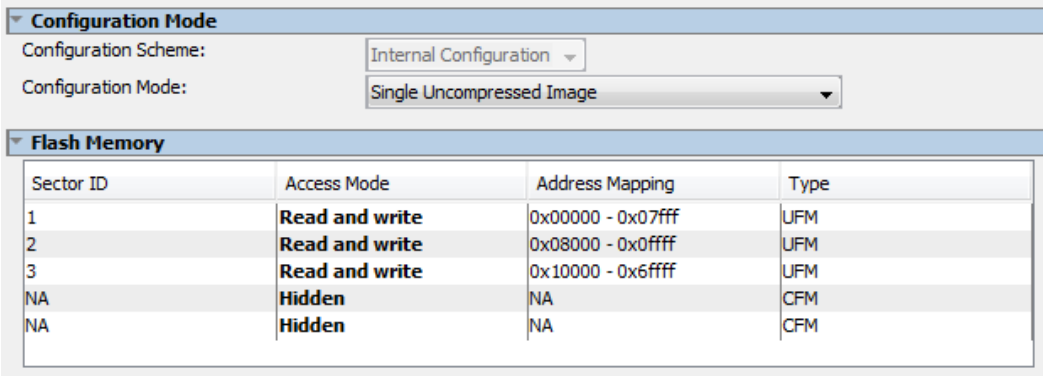
As explained previously, how many sectors that are available as user flash memory is dependent on how the chip is configured (how the FPGA bitstream and possibly other information will be saved into the on-chip flash). What this essentially comes down to is whether or not you're using dual configuration images, compressing the configuration image, or providing a memory initialization file alongside your configuration image. Having a compressed configuration image with no memory initialization provides you with the largest amount of user flash memory (only one sector is used for configuration, so the other four can be used as user flash). Refer to the table below to find out how many sectors are available for your application if you are using the flash or analog Max 10 variants:

Configuration	UFM1	UFM0	CFM2	CFM1	CFM0
Dual compressed images	UFM space	UFM space	—	—	—
Single uncompressed image	UFM space	UFM space	UFM space		—
Single compressed image	UFM space	UFM space	UFM space	UFM space	
Single uncompressed image with memory initialization	UFM space	UFM space	—	—	—
Single compressed image with memory initialization	UFM space	UFM space	—	—	—

If you are using the compact Max 10 variant, use the following table:

Configuration	UFM1	UFM0	CFM2	CFM1	CFM0
Dual compressed images	Not available				
Single uncompressed image	UFM space	UFM space	—	—	—
Single compressed image	UFM space	UFM space	—	—	—
Single uncompressed image with memory initialization	Not available				
Single compressed image with memory initialization	Not available				

This configuration setting can be found both inside of the On-Chip Flash IP parameter settings from within Qsys as well as inside of Quartus (make sure to set both settings to the same thing). To access the configuration settings from within Quartus, click on the following: Assignments->Device…, Device and Pin options…, and then click on “Configuration” in the sidebar. Within the On-Chip Flash IP parameter settings within Qsys, make sure to set the correct access modes for each sector. For most applications, all CFM sectors should be set to “Hidden” (so the software doesn’t accidentally modify the boot images) while the UFM sectors should be set to “Read and write.” When setting a sector to “Hidden” it won’t appear within the address mapping. The configuration used in this design example, is shown below:



Restrictions

There are two restrictions to keep in mind when using this design example. First off, when using Max 10 devices with larger amounts of flash memory (and in turn, larger page sizes) you might need to increase the amount of memory in your system to be able to hold an entire page's worth of data within memory. For instance, a 10M50 device has a page size of 8KB which means you need to have enough memory to store an 8KB buffer. For on-chip memory, this could be an issue if the rest of your software takes up a lot of space. When choosing off-chip memory, keep in mind how much space the page buffer will take up (assuming you need read-modify-write functionality when accessing flash).

Second, running this example requires a device with enough memory to store the entire regular C library. This is because the example implements a small user interface for accessing flash memory which requires use of the `scanf` line of input functions (which are stripped out in the "small" C library). If you have a device that doesn't have enough memory for the entire design example, all of the functions that work with the flash (`ReadByte()`, `WriteByte()`, and `EraseBlock()`) will still work correctly. Feel free to just copy those functions into your design or use them as a base for your own code.

How to Compile the Hardware

Follow the steps on the Design Store web page to extract and install the UserFlash platform file. The following steps describe how to setup a project in the Quartus II software in order to program the MAX10 FPGA device with the User Flash Memory design.

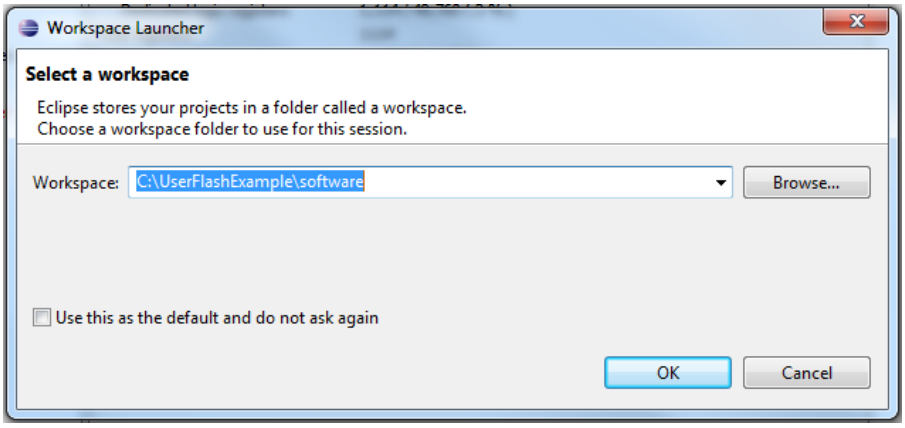
1. Launch Quartus II and open the top.qpf project file using File->Open Project.
2. Compile the project by clicking on Processing->Start Compilation.
3. Once the project is done compiling, launch the Quartus II Programmer from the tools menu (Tools->Programmer).
4. Plug in your development kit and make sure USB-Blaster II is shown next to the "Hardware Setup…" button. If it doesn't, click on that button and select the correct programmer.
5. Once the programmer is selected, click on "Auto-Detect" in the programmer's sidebar. If a dialog box appears asking for which device is on the JTAG chain, select "10M50DA" (assuming you have a 10M50DA device).
6. Double click in the File list where it says "<none>" and select the output_files/UserFlash.sof file.
7. Click "Start" to program the device.

How to Compile the Software

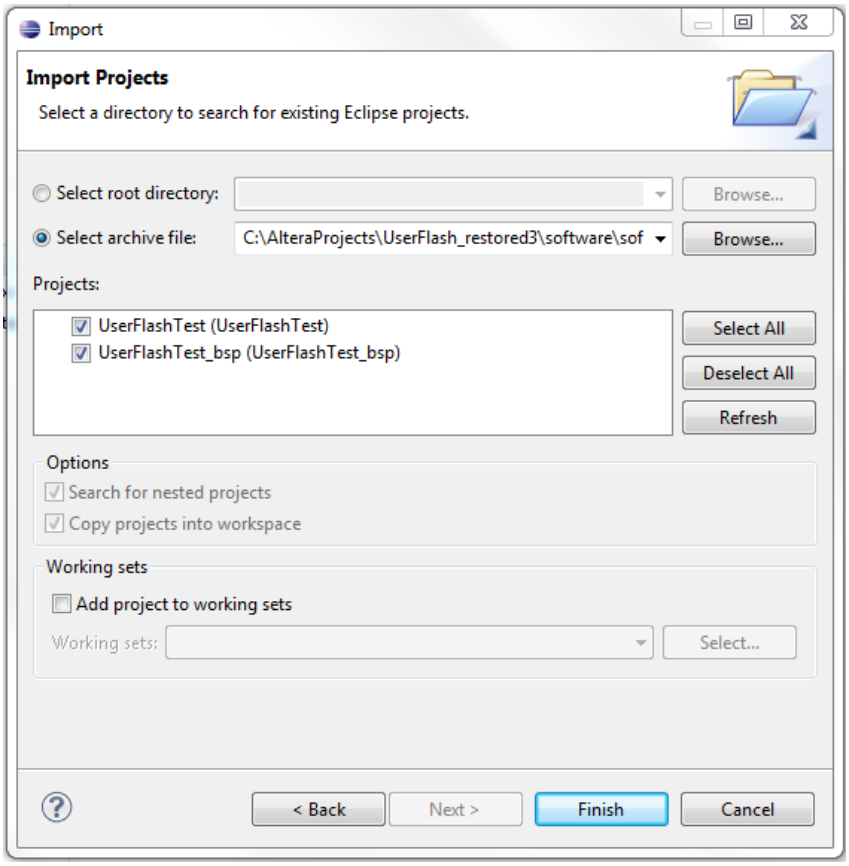
The following steps will walk you through restoring the included software project and programming it to the device. This software contains both an application project and a BSP project meant to run on a 10M50 device. The meat of the example is contained within the UserFlash.c file inside of the UserFlashTest application project.

1. Open up the Nios II Software Build Tools by clicking on Tools->Nios II Software Build Tools for Eclipse.
2. When the workspace selection window appears, browse to <design-example-installation-directory>\software and hit OK. For instance, if you extracted the example into

C:\UserFlashExample, then the following image shows where your workspace will be located:



3. Next, go to File->Import… and then select General->Existing Projects into Workspace. Click Next.
4. At this window, select the "Select archive file" option and browse to the "softwarearchive.zip" file within the software folder. Click Finish.



5. Once the projects have been imported you can open up the example code by opening the UserFlash.c file within the UserFlashTest application project.
6. Build the project by going to Project->Build All
7. To run the example, you need to set up a run configuration. Click on Run->Run Configurations. Next, right-click on "Nios II Hardware" in the sidebar and select "New." Under Project name, select the UserFlashTest project. Next, click on the "Target Connection" tab and hit "Refresh

Connections” if the Max 10 device doesn’t show up (you may have to scroll to the right to see that button). If no connections appear, make sure the board is both plugged in and already programmed with the .sof file as described in the previous section. Once that’s connected, click on Apply and then Run.