ALTERA®

White Paper

Configuring the MicroBlaster Fast Passive Parallel Software Driver

Introduction

The MicroBlaster fast passive parallel (FPP) software driver is an embedded solution for configuring Altera® SRAM-based programmable logic devices (PLDs). The FPP configuration scheme allows eight bits of configuration data to be loaded to the target device on every clock cycle, significantly reducing configuration time.

You can customize the modular source code's I/O control routines (provided as a separate file) for your system. The MicroBlaster FPP software driver supports raw binary file (**.rbf**) format generated by the Quartus® II software, and was developed and tested on the Windows NT platform. The file size of this Windows NT driver is about 40 Kbytes.

This white paper describes the source code of MicroBlaster FPP driver and explains how it can be ported to other embedded platforms.

I/O Pin Assignments

Because the writing and reading of data to and from the I/O ports on other platforms maps to the parallel port architecture, this document focuses on the assignment of FPP configuration signal pins to a parallel port. These pin assignments reduce the required modification on the original source code. Table 1 shows the assignment of the FPP configuration signals to a parallel port.

Table 1. FPP Configuration Signal Pin Assignments to a Parallel Port

Port	Bit *(1)*							
	7	6	5	4	3	2	1	0
Port 0 *(2)*	DATA7	DATA6	DATA5	DATA4	DATA3	DATA2	DATA1	DATA0
Port 1 *(2)*	CONF_DONE	-	-	nSTATUS	-	-	-	-
Port 2 *(2)*	-	-	-	-	nCONFIG	-	-	DCLK

Notes to Table 1:
(1) Bit 7 is the most significant bit (MSB).
(2) This port refers to the index from the base address of the parallel port (e.g., 0x378).

WP-MCRBLSTRPLL-1.0

The Interface

The MicroBlaster FPP software driver's source code consists of two modules: data processing and I/O control. The data processing module reads the programming data from the RBF, rearranges it, and sends it to the I/O control module. The I/O control module then transfers the configuration data to the target PLD. Periodically, the I/O control module monitors certain configuration pin's status to see if any errors have taken place during the configuration process. When an error occurs, the MicroBlaster FPP source code re-initiates the configuration process.

Source File

Table 2 describes the MicroBlaster FPP source files.

Table 2. MicroBlaster FPP Source Files

File	Descriptions
mblaster_fpp.c	Contains the `main()` function. It manages the processing of the programming input file, instantiates the configuration process, and responds to any configuration errors. This file is platform independent.
mb_fpp_io.c **mb_fpp_io.h**	These files handle the I/O control functions and are platform dependent. They are developed for PCs only running Windows NT. Modify these files to support other platforms.

Constant

The source code contains several program and user-defined constants. While the program constant should not be changed, user-defined constants can be changed if necessary. Table 3 summarizes the constants.

Table 3. Program & User-Defined Constants

Constant	Type	Description
`WINDOWS_NT`	Program	Windows NT operating system
`EMBEDDED`	Program	Embedded microprocessor system or other operating system
`PORT`	Program	Determines the operating platform
`nSTATUS`	Program	`nSTATUS` signal (Port 1, Bit 4)
`CONF_DONE`	Program	`CONF_DONE` signal (Port 1, Bit 7)
`INIT_CYCLE`	User	The number of clock cycles to toggle after configuration is done to initialize the device. Each device family requires a specific number of clock cycles.
`RECONF_COUNT_MAX`	User	The maximum number of auto-reconfiguration attempts allowed when the program detects an error.
`CHECK_EVERY_X_BYTE`	User	Check `nSTATUS` pin for every `X` number of bytes programmed. Do not use `0`.
`CLOCK_X_CYCLE (optional)`	User	The number of additional clock cycles to toggle after `INIT_CYCLE`. Use `0` if no additional clock cycles are required.

Global Variables

Table 4 summarizes the global variables used when reading or writing to I/O ports. Map the I/O ports of your system to these global variables.

Table 4. Global Variables

Global Variables	Type	Descriptions
`sig_port_maskbit[X]`	Integer array	Holds the mask bit position for `nSTATUS` and `CONF_DONE` signals. X = 0 refers to `nSTATUS` X = 1 refers to `CONF_DONE`
`port_reset[Z]`	Integer array	Holds the reset value of the output ports. Z = 0 refers to `Port 0` Z = 1 refers to `Port 1`

Functions

Table 5 describes the parameters and the return value of some of the functions in the source code. Only functions declared in the **mb_fpp_io.c** file are discussed, because you need to customize these functions to work on platforms other than Windows NT. These functions contain the I/O control routines.

Table 5. I/O Control Functions

Functions	Parameters	Return Value	Descriptions
`ReadPort`	int port	Integer	This function reads the value of the port and returns it. Only the least significant byte contains valid data. *(1)*
`FlushPort`	int port int data	None	This function writes the data to the port. Data of integer type is passed to the function. Only the least significant byte contains valid data. `FlushPort` function can access either `DATA` port or `CONTROL` port. *(1)* The `DATA` port carries the byte-wide configuration data whereas `CONTROL` port gives access to control signals (i.e., `DLCK` and `nCONFIG`).
`Dump2PortIO`	int port int data	None	This function passes the configuration data to `FlushPort` function. Data is then clocked continuously to the device.

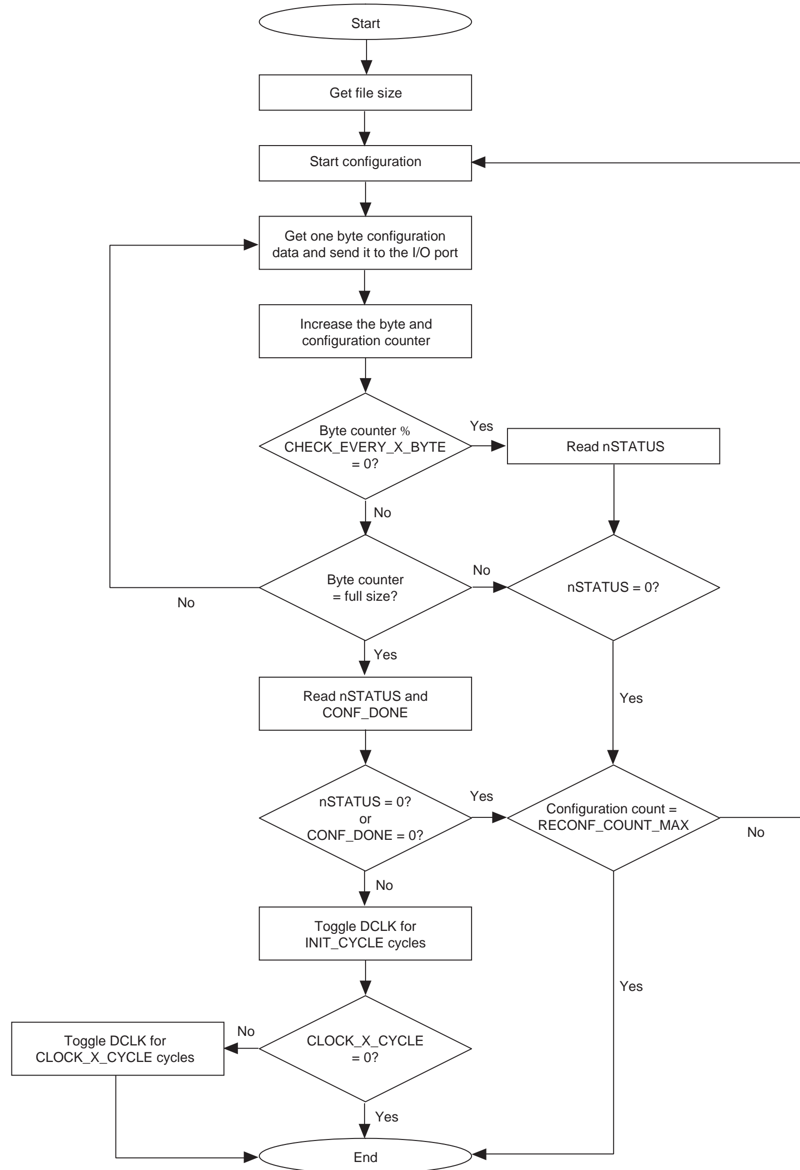
Note to Table 5:

(1) This port refers to the index from the base address of the parallel port (e.g., 0x378).

Program Flow

Figure 1 illustrates the program flow of the MicroBlaster FPP software driver. The CHECK_EVERY_X_BYTE, RECONF_COUNT_MAX, INIT_CYCLE, and CLOCK_X_CYCLE constants determine the flow of the configuration process. See Table 3 for the definition of these constants.

Figure 1. MicroBlaster FPP Program Flow



Porting

Three platform-dependent routines handle the read and write operations in the I/O control module. The read operation reads the value that is sent to the input port. The write operation writes data to the output port.

To port the source code to other platforms or embedded systems, you must implement your I/O control routines in the I/O control functions (i.e., `ReadPort`, `FlushPort`, and `DumptoPortIO`). See Table 5. You can modify the existing code to suit your applications. Your I/O control routines can be implemented between the following compiler directives:

```
#if PORT == WINDOWS_NT
/* original source code */
#else if PORT == EMBEDDED
/* put your I/O control routines source code here */
#endif
```

Reading

The `ReadPort` function accepts the port as an integer parameter and returns an integer value. Your code should map or translate the port value defined in the parallel port architecture (see Table 1) to the I/O port definition of your system.

For example, when reading from port 1, your source code should read the `CONF_DONE` and `nSTATUS` signals from your system (defined in Table 1). The code should then rearrange these signals within an integer variable so that the values of `CONF_DONE` and `nSTATUS` are represented by bit position 7 and 4 of the integer, respectively. This behaviorally maps your system's I/O ports to the pins in the pin assignments of the parallel port architecture. By adding these lines of translation code to the **mb_fpp_io.c** file, you can avoid modifying code in the **mblaster_fpp.c** file.

Writing

There are two functions governing the write operation: `Dump2PortIO` and `FlushPort`. The `Dump2PortIO` function receives two integer parameters (port and data) from the **mblaster_fpp.c** file. This "port" refers to the index from the base address of the parallel port (e.g., 0x378 whereas "data" refers to control signal or configuration data.

If a control signal reaches `Dump2PortIO` (e.g., `nCONFIG`), it will be sent to the `FlushPort` function to be driven out of the I/O port. However, if the `Dump2PortIO` function receives configuration data (`DATA[7..0]`) that requires continuous clocking, one clock cycle (`DCLK`) will also be driven out for each byte of configuration data.

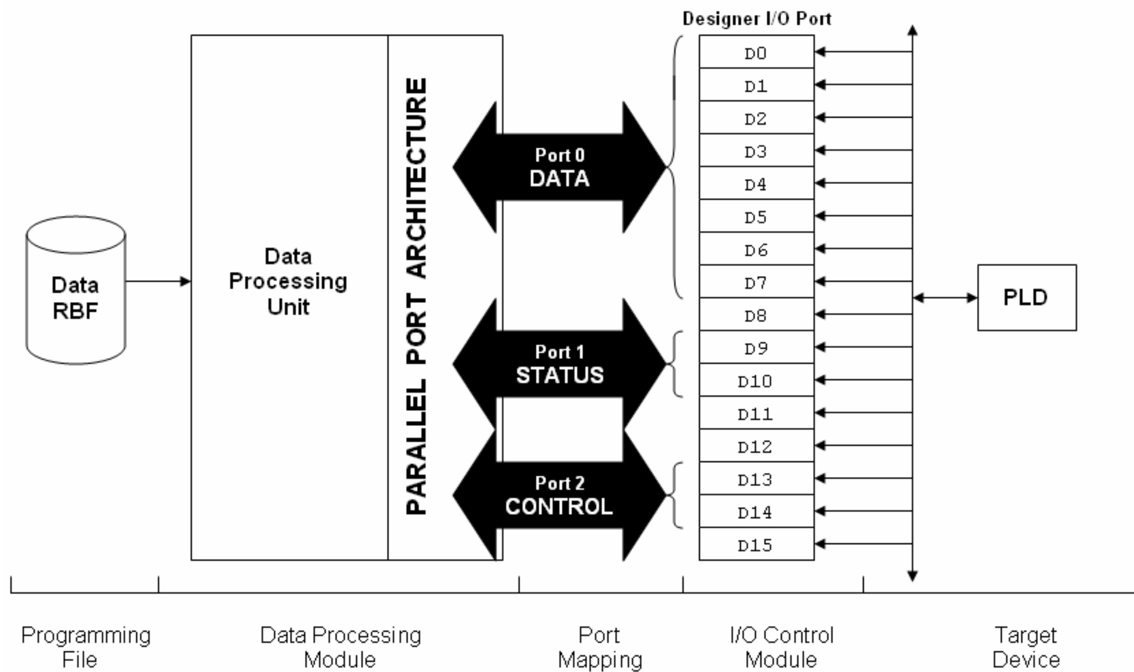
Due to parallel port architecture limitations, the `Dump2PortIO` function takes three steps to drive one byte of configuration data and one clock cycle out of the I/O port. However, an embedded system can reduce this to two steps, with the condition that it has more than eight bits of I/O port registers.

You can modify the `FlushPort` function the same way as the `ReadPort` function. Your code maps or translates the port value that is defined in the parallel port architecture (see Table 1) to the I/O port defined by your system. By adding new I/O port translation codes to the **mb_fpp_io.c** file, you can avoid modifying code in the **mblaster_fpp.c** file.

Example

Figure 2 shows an embedded system holding 12 configuration signals in the data registers of an embedded 16-bit microprocessor. When reading from the I/O ports, the I/O control routine reads the values of the data registers and maps them to the particular bits in the parallel port registers (`Port 0` to `Port 2`). These bits are later accessed and processed by the data processing module.

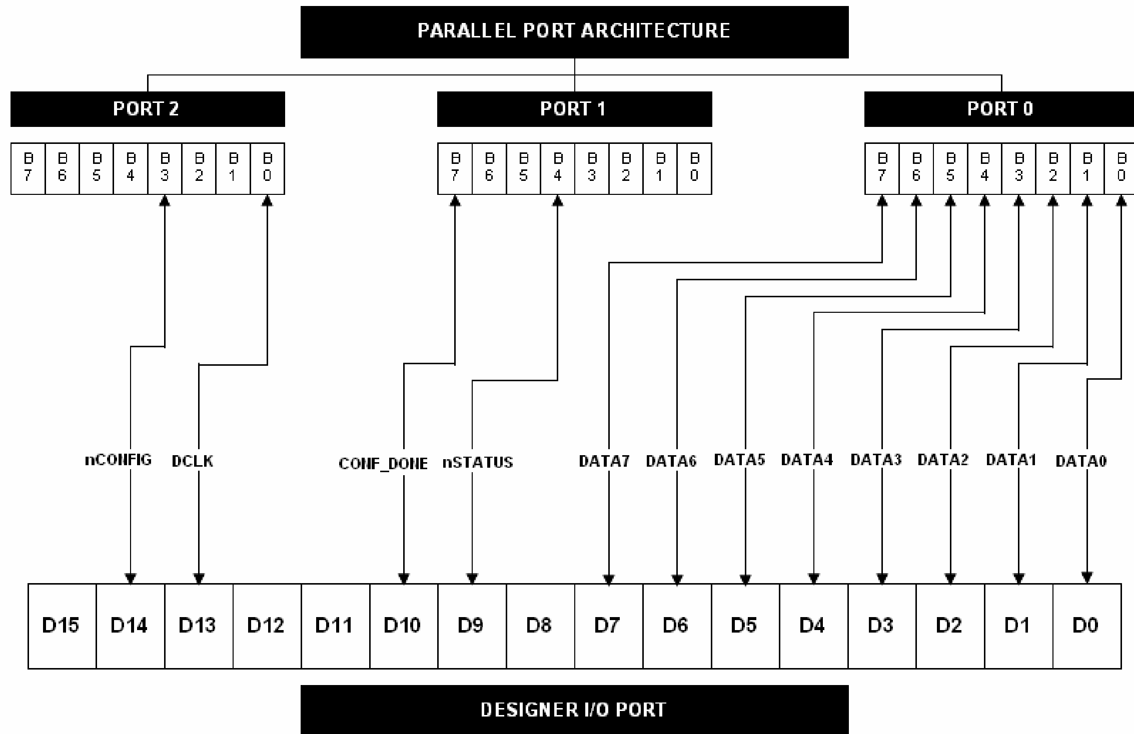
Figure 2. Example of I/O Reading & Writing Mapping Processes



When writing, the values of the signals are stored in the parallel port registers (`Port 0` to `Port 2`) by the data processing module. The I/O control routine then reads the data from the parallel port registers and sends it to the corresponding data registers of the microprocessor.

Figure 3 shows I/O port mapping.

Figure 3. I/O Port Mapping



Conclusion

The MicroBlaster FPP embedded configuration source code is modular so you can easily port it to other platforms. It offers a simple, inexpensive embedded system to accomplish FPP configuration for Altera PLDs.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com

Copyright © 2003 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries.* All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.