BUTTERFLY*LABS*

BitFORCE SC Communication Protocol
Revision 1.0.0 – DRAFT

WARNING:

This document is in draft stage. It will be subject to changes until preliminary and final revisions are published.

December 2012

SecureHASH:
b9a199a934c5522bc00f97b2cce74ec69dc642abe1b95e5585b4816f4c04d5cb

1. Introduction

The purpose of this document is to examine the communication protocol between a host device, such as a PC, Mac or any device capable of operating as a host and use FTDI driver to communicate with BitFORCE SC series.

It must be noted that the BitFORCE SC uses FTDI USB engine to establish connection with its host, and thus having the FTDI drivers installed on the system is mandatory. Otherwise, the host may not recognize the device.

FTDI Drivers are available at http://www.ftdichip.com/Drivers/VCP.htm

2. Backward Compatibility

The BitFORCE SC devices are generally not compatible with the older FPGA family of products. The only exception is the "Device Identification" command, as this command remains valid with the new series of BitFORCE SC. Please refer to the "Commands" section of this document.

3. Enumeration

Once the BitFORCE device is connected to host, it will be enumerated as a "COM" port on Windows ® Based machines and as a "ttyusb" (under **/dev/**) on UNIX/LINUX based operating systems. Should the OS fail recognition, the installation of FTDI drivers must be verified.

BitFORCE SC devices are operational on start. The only exception is the MiniRig SC products which may take a few hundred milliseconds before they are fully up. This would normally not be sensed by the user or the OS, as it does not interfere with normal enumeration on the host.

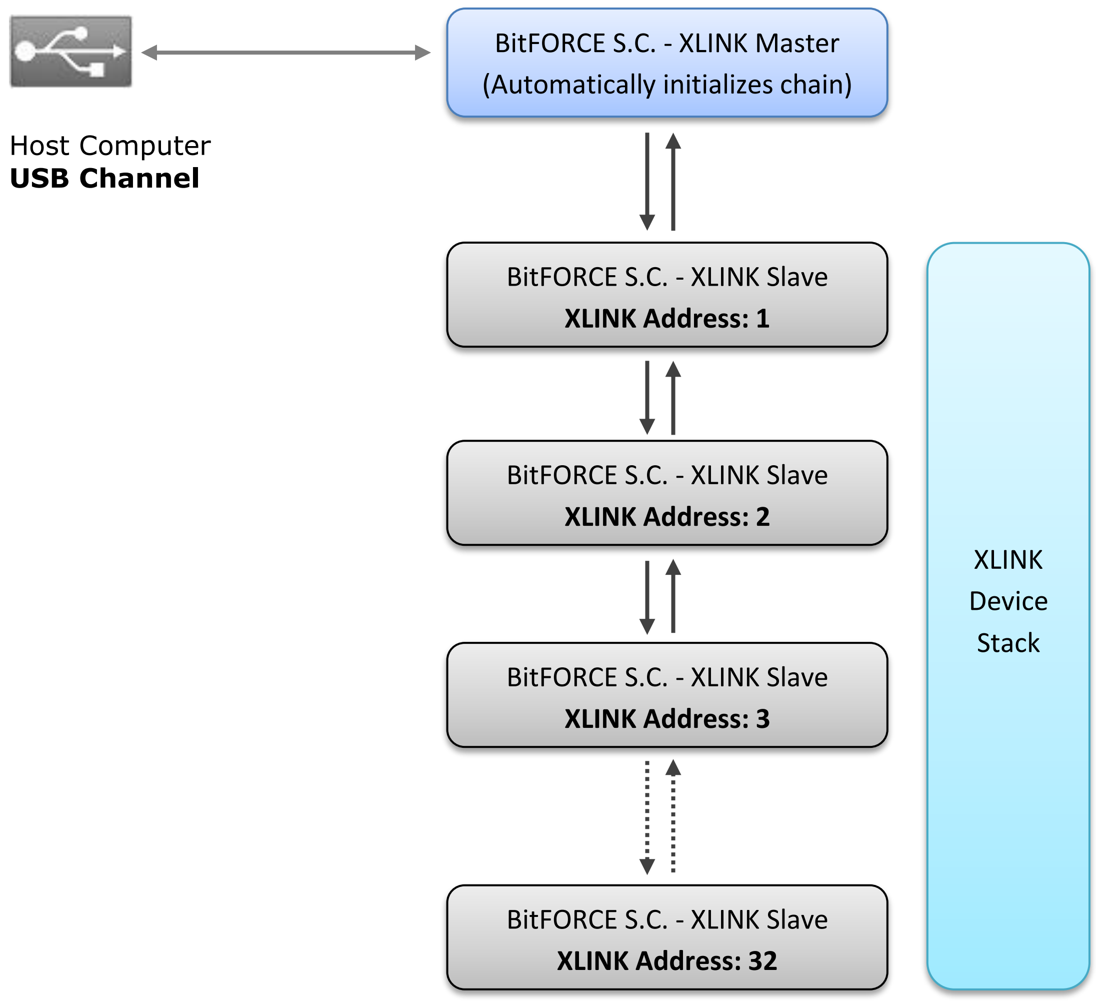
Once enumerated, the device can be detected in host software by issuing a simple "Device Identification" command (later explained in this document). The device will respond with its name and version, identifying itself as a BitFORCE SC device.

4. MiniRig XLINK Chain

The MiniRig systems may incorporate more than twenty cards inside the enclosure. One of the cards which is connected to the USB will operate as XLINK chain master, and all other cards which are connected to the master card through XLINK cables, will operate as slaves.

Upon power-up, the master card will automatically initialize all devices in the chain. This will be transparent to the user or the host, even though it may take a few hundred milliseconds for the master card to be fully operational and responsive to the USB commands.

It must be noted that the data will pass each card before reaching the next card down in the chain. As a result, should one card electronically fail, all devices under it will no longer respond to chain commands. This anomaly can be simply detected by asking the master to report the number of devices in the chain.


BitFORCE S.C. - XLINK Master
(Automatically initializes chain)
Host Computer
USB Channel
BitFORCE S.C. - XLINK Slave
XLINK Address: 1
BitFORCE S.C. - XLINK Slave
XLINK Address: 2
BitFORCE S.C. - XLINK Slave
XLINK Address: 3
BitFORCE S.C. - XLINK Slave
XLINK Address: 32
XLINK Device Stack

The figure above illustrates XLINK device stack, implemented in MiniRig series.

5. Sending command to devices

When sending commands, we have two possible targets: (a) Master device (including Singles and Jalapeños) and (b) Slave devices in the MiniRig chain.

(a) Commands sent to Master device (including Singles and Jalapeños) are straightforward. The three characters of the command (ZGX, ZTX, ZDX, etc.) can be sent directly to device via USB channel. No special treatment is required.

(b) Commands need to be sent to a device in chain must include a **"@XY" tag** before the actual command. The "X" is a **byte**, defining the target address (can range from 1 to 32 in binary format). "X" being "1" asks the Master to forward the command to the first device in chain after the master, "2" to the second device in chain after the master and so on.

The "Y" is a **byte,** indicating the size of the stream after this three-character header.

For instance, to send a “ZGX” command to the 4th device in the chain after the master, the command output will look like {64, 4, 3, 'Z', 'G', 'X'} (Note that “64” is the ASCII number for '@' character, “4” means the 4th device after the master, “3” means three bytes to deliver, which is three for the 'ZGX').

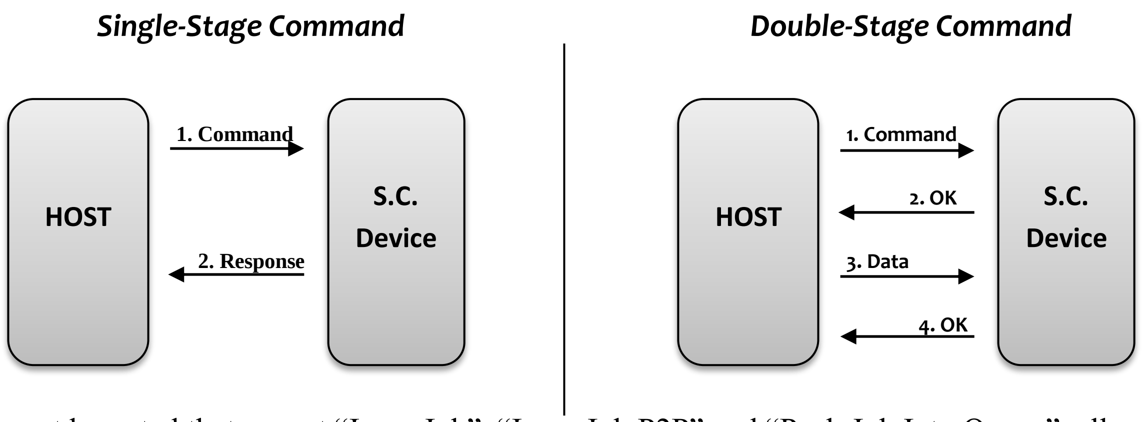
The following 'C' structure clarifies the structure of XLINK forward command:

```
struct DataForwardToChain
{
    unsigned __int8 Header;        // '@' by default, or numeric 64
    unsigned __int8 DeviceAddress; // Address of the device in chain
    unsigned __int8 PayloadSize;   // number of bytes to deliver
    unsigned __int8 PayloadData[]; // Bytes to be delivered, 'ZGX' for instance
                                   // Maximum allowed is 255 bytes.
};
```

6. Type of commands

Commands are either single-stage or double-stage. All the commands except job-issuing ones are single-stage; meaning device only responds to the command and goes back to IDLE stage, ready for the next command to be issued. Both single-stage and two-stage commands are completely asynchronous.

Double-stage commands (such as job-issuing) require a second input after the primary command is issued. The chart below illustrates the difference:



It must be noted that except “Issue Job”, “Issue Job P2P” and “Push Job Into Queue”, all commands are single-staged.

7. Invalid Commands & Timeouts

Should the host issue an unrecognizable command or stream of data to the device, the device will either respond with “ERROR: UKNOWN” or will not respond at all for duration of 1ms. This is to prevent device from dead-locking due to bugs in the operating software on the host.

For the double-stage commands, upon each stage, the device will await new data for duration of approximately **400us**. Should the host fail to provide data during this time span, the device will automatically ignore the issued command altogether and will return to IDLE state.

In any case, after a period of 1ms, the device will have returned to IDLE state and will be ready to accept new commands. This can be considered as an error-recovery strategy.

8. BitFORCE S.C. Commands

Below you will find the list of commands issuable BitForce SC devices. Note that all of these commands can be issue to slave cards in MiniRigs as well (using the XLINK forward tag).

8.1. 'ZGX' – Device Identification

This is a single stage command. Once issued to the device, it will respond with 'BitFORCE SC X.Y" with X and Y being numeric values defining the version of the device. By default, X is '1' and Y is '0'.

8.2. 'ZCX' – Device Information

This is a single stage command. Once issued to the device, it will return basic information about itself:

```
DEVICE: BitFORCE SC
FIRMWARE: <Version of the firmware, starting with 1.0.0>
ENGINES: <Number of ASIC engines installed)
XLINK MODE: <MASTER or SLAVE>
XLINK PRESENT: <YES or NO>
DEVICES IN CHAIN: <Count of slave devices in XLINK chain> [CONDITIONAL]
OK
```

The 'XLINK MODE' tells whether the device is the XLINK master or a simply an XLINK slave in the chain.

The 'DEVICES IN CHAIN' field above will only appear if 'XLINK PRESENT' is valid. Should the XLINK not be present (the XLINK chip not installed on board), this field will not appear in the output.

Please note that the end of each line is marked with a CrLf (i.e. **\n**).

8.3. 'ZMX' – Blink

This is a single stage command. Once issued to the device, it will respond with 'OK' immediately and blink for approximately two seconds. This can be used for visually identifying the device.

8.4. 'ZTX' – Get Voltages

This is a single stage command. Once issued to the device, it will respond with a three field text, containing the on-board voltages in millivolts for 1.0V, 3.3V and VccMain (either 12V for Single and MiniRig cards or 4.5V to 5.0V for Jalapeno devices):

```
3290,1001,12400
```

The first field is 3.3V, second is 1.0V and the third is the Vcc-Main. Please note that this line is terminated with a CrLf.

8.5. 'ZLX' – Get Temperatures

This is a single stage command. Once issued to the device, it will respond with detected temperature of two temperature sensors on board:

```
Temp1: 29, Temp2: 28
```

Please note that this line is finished with a CrLf. This command does not exist in FPGA series of products.

8.6. 'ZJX' – Get Firmware Version

This is a single stage command. Once issued to the device, it will respond with the actual firmware version of the device:

```
1.0.0
```

This is an example. Actual value will depend on the firmware version of the device. Please note that this line is finished with a CrLf.

8.7. 'ZFX' – Get Job Result Status

This is a single stage command. Once issued to the device, it will respond with the status of the engines. It can be either one of the below:

- BUSY
- IDLE
- NO-NONCE
- NONCE-FOUND:<XXXXXXXX>,<YYYYYYYY>,<ZZZZZZZZ>…

To further explain these possible responses:

-- BUSY: Device is processing a nonce
-- IDLE: Means the device has never processed any jobs and is not processing one.
-- NO-NONCE: Device has finished processing the job and no nonces were found
-- NONCE-FOUND: Means nonces have been detected. Nonces are separated with a colon and they are formatted as 32Bit hexadecimal numbers (i.e. 3ABEF01E, 2A0000B9, etc).

A true-world example for the Nonce-Found:

```
'NONCE-FOUND:0F1A9E1F,0200B4A0,89B67A00'
```

NOTE: The response will be terminated with a CrLf.

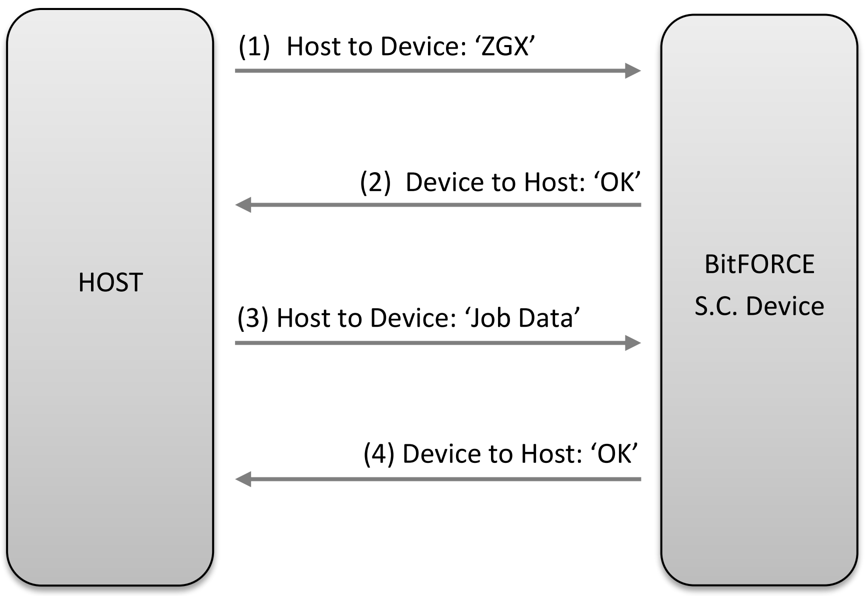
8.8. ‘ZDX’ – Handle Full Range Job (Double Stage)

This is a **double stage** command. Once issued to the device, it will respond with ‘OK’. This means the device is waiting for job data to be sent by the host. If the job data is not received within 400us, the device assumes an anomaly exists and will respond with ‘ERR: TIMEOUT’ while returning to IDLE state and waiting for a new command to be issued.

Under normal circumstances however, the host will send the job information and once received by the device (assuming the data is correctly formatted), will respond with ‘OK’. This means the device has started processing the job.

From this point, the status of engines can be determined by issuing a single ‘Get Job Result Status’ command (see 8.7).

The diagram below illustrates this command:

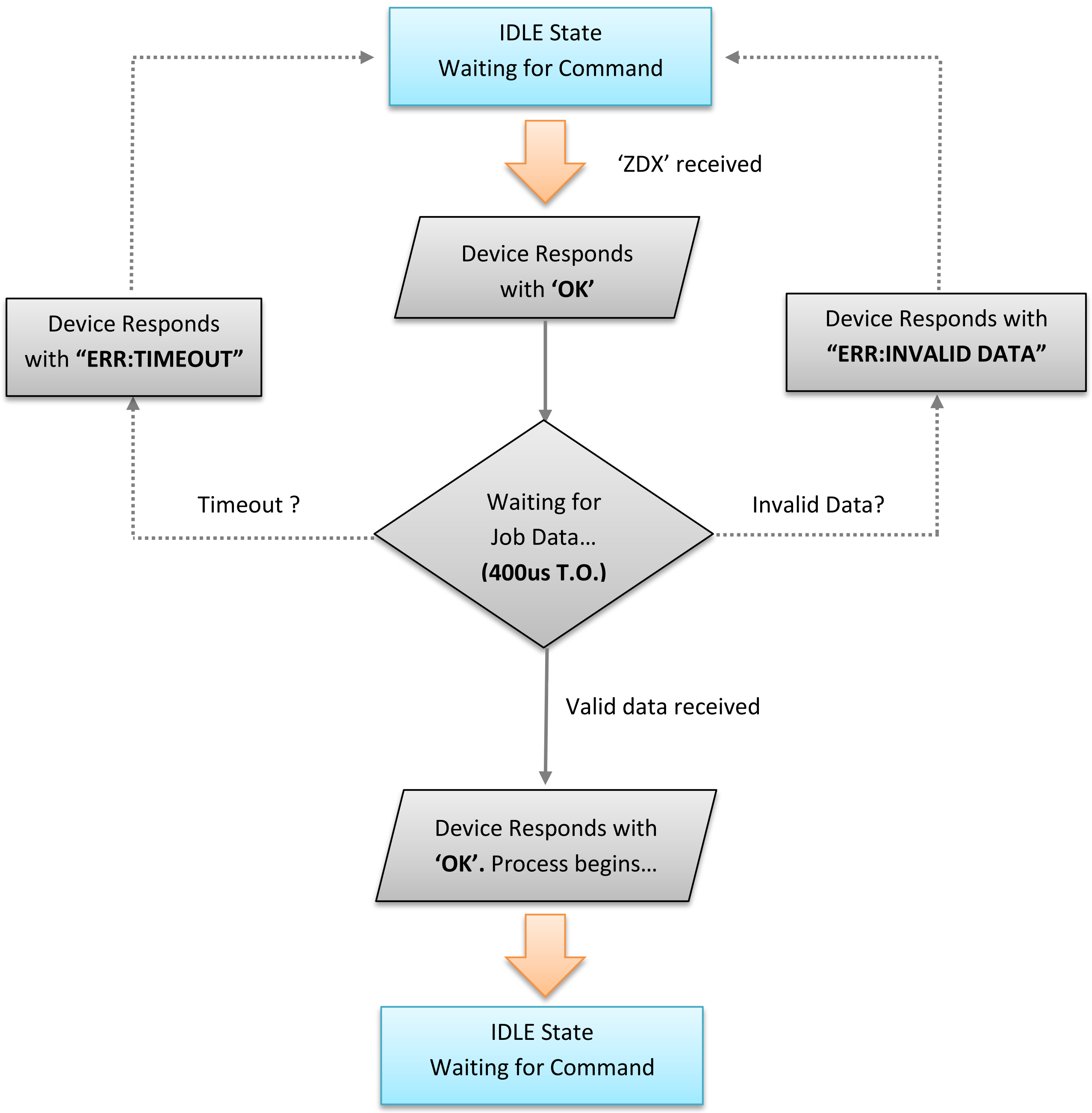


It must be noted that the device will wait 400us for the host to send job data. After this period, the device will send an **‘ERR: TIMEOUT’** message to host and the operation will be aborted. Should the device receive data but the format is incorrect, it will respond with **‘ERR: INVALID DATA’** and aborts the operation, returning to IDLE state.

Format of the Full-Range Job data is described in the ‘C’ structure below:

```
struct FullNonceRangeJob
{
    unsigned __int8 dataSize;       // Payload length, which is 45 (32+12+1)
    unsigned __int8 midState[32];   // 256Bits of midstate data
    unsigned __int8 blockData[12];  // 12 bytes of merkel data
    unsigned __int8 endOfBlock;     // Should be 0xAA
};
```

It must be noted that the device will not notify the host once the process has finished. The host must implement a polling strategy, asking the unit whether it has finished processing or not every 5ms or so. Job-issue commands are fully asynchronous and the device **will not be blocked** until job-processing is finished.


IDLE State
Waiting for Command
'ZDX' received
Device Responds with **'OK'**
Device Responds with **"ERR:TIMEOUT"**
Device Responds with **"ERR:INVALID DATA"**
Timeout ?
Waiting for Job Data... **(400us T.O.)**
Invalid Data?
Valid data received
Device Responds with **'OK'.** Process begins...
IDLE State
Waiting for Command

The figure above illustrates how 'ZDX' command operates.

8.9. ‘ZPX’ – Handle Custom Range Job (Double Stage)

This command is identical to Full-Range Job Issue (ZDX), except that the nonce-range is included in the data structure:

```
struct CustomNonceRangeJob
{
    unsigned __int8  dataSize;      // Payload length = 53 (32+12+4+4+1)
    unsigned __int8  midState[32];  // 256Bits of midstate data
    unsigned __int8  blockData[12]; // 12 bytes of merkel data
    unsigned __int32 nonceBegin;    // 4 Bytes of Nonce-Begin
    unsigned __int32 nonceEnd;      // 4 Bytes of Nonce-End
    unsigned __int8  endOfBlock;    // Should be 0xAA
};
```

Once the command is successfully issued, the device will start processing the job, taking the ‘`nonceBegin`’ and ‘`nonceEnd`’ into consideration. Host can call the usual ‘Get Job Result Status’ command (see 8.7) to learn the status of job in process.