Microsemi makes no warranty, representation, or guarantee regarding the information contained herein or the suitability of its products and services for any particular purpose, nor does Microsemi assume any liability whatsoever arising out of the application or use of any product or circuit. The products sold hereunder and any other products sold by Microsemi have been subject to limited testing and should not be used in conjunction with mission-critical equipment or applications. Any performance specifications are believed to be reliable but are not verified, and Buyer must conduct and complete all performance and other testing of the products, alone and together with, or installed in, any end-products. Buyer shall not rely on any data and performance specifications or parameters provided by Microsemi. It is the Buyer’s responsibility to independently determine suitability of any products and to test and verify the same. The information provided by Microsemi hereunder is provided “as is, where is” and with all faults, and the entire risk associated with such information is entirely with the Buyer. Microsemi does not grant, explicitly or implicitly, to any party any patent rights, licenses, or any other IP rights, whether with regard to such information itself or anything described by such information. Information provided in this document is proprietary to Microsemi, and Microsemi reserves the right to make any changes to the information in this document or to any products and services at any time without notice.

About Microsemi

Microsemi, a wholly owned subsidiary of Microchip Technology Inc. (Nasdaq: MCHP), offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions, security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Learn more at www.microsemi.com.
# Contents

1 Revision History ................................................................. 1  
1.1 Revision 6.0 ................................................................. 1  
1.2 Revision 5.0 ................................................................. 1  
1.3 Revision 4.0 ................................................................. 1  
1.4 Revision 3.0 ................................................................. 1  
1.5 Revision 2.0 ................................................................. 1  
1.6 Revision 1.0 ................................................................. 1  

2 PolarFire FPGA Programming ...................................................... 2  
2.1 Bitstream Generation .......................................................... 4  
2.1.1 Bitstream Generation Flow .............................................. 4  
2.1.2 Adding sNVM Data to the Bitstream .................................. 5  
2.1.3 Adding User Security Settings to the Bitstream ................. 5  
2.1.4 Configuring Bitstream Components ................................. 9  
2.1.5 Programming File Size ................................................ 10  
2.2 Device Programming Flow .................................................. 11  
2.2.1 Programming Time ..................................................... 12  
2.3 Programming Modes ....................................................... 12  
2.3.1 JTAG Programming .................................................... 12  
2.3.2 SPI Slave Programming ............................................... 17  
2.3.3 SPI Master Programming ............................................. 20  
2.4 Bypassing the Back Level Protection ................................... 36  
2.4.1 Bypass Back Level Protection Use Case .......................... 37  
2.5 PolarFire FPGA I/O States During Programming ................... 37  
2.6 Programming Recommendations ......................................... 38  
2.7 Brownout During Programming ......................................... 38  
2.8 Zeroization ................................................................. 39  

3 Programming the External SPI Flash .......................................... 40  
3.1 Supported SPI Flash Devices ............................................. 40  
3.2 SPI Directory .............................................................. 40  
3.3 Use Models for Programming SPI Flash ................................ 41  
3.3.1 Programming the SPI Flash Using External Processor ........ 41  
3.3.2 Programming the SPI Flash Using JTAG ......................... 42  
3.3.3 Programming the SPI Flash Using Fabric User Logic .......... 44  

4 Appendix: Error Codes .......................................................... 45
## Figures

| Figure 1 | PolarFire FPGA Programming Modes | 2 |
| Figure 2 | PolarFire Device Programming Modes and Interfaces | 3 |
| Figure 3 | Bitstream Generation in Libero Design Flow | 4 |
| Figure 4 | Design and Memory Initialization | 5 |
| Figure 5 | Configure Security—Custom Security Options | 6 |
| Figure 6 | Configure Security Wizard—Update Policy | 7 |
| Figure 7 | Configure Programming Options | 8 |
| Figure 8 | Configure Bitstream Window | 9 |
| Figure 9 | PolarFire FPGA Programming Flow | 11 |
| Figure 10 | JTAG Signals Timing Diagram | 13 |
| Figure 11 | JTAG Programming Using External Programmer | 13 |
| Figure 12 | Device Programming in JTAG Chain | 14 |
| Figure 13 | Connecting FlashPro Programmer to a PolarFire FPGA | 15 |
| Figure 14 | Programming Using External Microprocessor | 16 |
| Figure 15 | SPI Slave Programming Using External Programmer | 18 |
| Figure 16 | SPI Slave Programming Using External Microprocessor | 19 |
| Figure 17 | Recommended Board Configuration for SPI Master Programming | 20 |
| Figure 18 | Core System Services IP Interfacing with Fabric User Logic | 22 |
| Figure 19 | Core System Services Configurator | 23 |
| Figure 20 | Auto Update High-Level Flowchart | 25 |
| Figure 21 | Auto Update Detailed Flow | 28 |
| Figure 22 | Auto Update Setting | 28 |
| Figure 23 | Design Version | 29 |
| Figure 24 | Ping Pong Use Model | 29 |
| Figure 25 | Golden Image Use Model | 30 |
| Figure 26 | Single Image Use Model | 31 |
| Figure 27 | IAP Flowchart | 34 |
| Figure 28 | Multi-Image Use Model | 35 |
| Figure 29 | Selecting Bypass Back Level Protection Feature | 36 |
| Figure 30 | Status of Bypass Back Level Protection | 36 |
| Figure 31 | I/O States During Programming (JTAG Mode Only) | 38 |
| Figure 32 | SPI Flash Directory | 40 |
| Figure 33 | SPI Flash Programming Using External Processor | 41 |
| Figure 34 | SPI Flash Programming Using JTAG | 42 |
| Figure 35 | SPI Flash Programming in Libero SoC | 42 |
| Figure 36 | Run PROGRAM_SPI_IMAGE Action | 43 |
| Figure 37 | SPI Flash Programming Using Fabric User Logic | 44 |
# Tables

| Table 1 | PolarFire FPGA Programming Interfaces and Bitstream Formats | 4 |
| Table 2 | Programming Files Sizes—Custom Security Disabled | 10 |
| Table 3 | Programming Files Sizes—Custom Security Enabled | 10 |
| Table 4 | JTAG Pins | 12 |
| Table 5 | FlashPro Header Signals | 14 |
| Table 6 | System Controller SPI Pins | 17 |
| Table 7 | System Controller's SPI Configuration | 18 |
| Table 8 | Device Program Initiation Sources | 20 |
| Table 9 | System Controller's SPI Configuration | 21 |
| Table 10 | System Service Descriptor | 21 |
| Table 11 | Bitstream Authentication Service Request | 24 |
| Table 12 | Bitstream Authentication Service Mailbox Format | 24 |
| Table 13 | IAP Image Authentication Service Request | 24 |
| Table 14 | Example Auto Update Conditions | 27 |
| Table 15 | Auto Update System Service Request | 27 |
| Table 16 | IAP Verify Request by Image Index | 32 |
| Table 17 | IAP Verify Request by Image Address | 32 |
| Table 18 | IAP Program Request by Image Index | 33 |
| Table 19 | IAP Request by Image Address | 33 |
| Table 20 | Mailbox Format | 33 |
| Table 21 | Bypass Back Level Protection Use Case | 37 |
| Table 22 | I/O States for Various Programming Modes | 37 |
| Table 23 | Error Codes | 45 |
1 Revision History

The revision history describes the changes that were implemented in the document. The changes are listed by revision, starting with the current publication.

1.1 Revision 6.0
Updated the document for Libero SoC v12.1.

1.2 Revision 5.0
The following is a summary of the changes made in this revision.

- Updated the document for Libero SoC PolarFire v2.3.
- The flash freeze functionality is de-featured.
- Added information about Bypass the back level protection feature. For more information, see Bypassing the Back Level Protection, page 36.
- Updated the information about zeroization. For more information, see Zeroization, page 39.
- Update the Programming file sizes table. For more information, see Table 2, page 10 and Table 3, page 10.

1.3 Revision 4.0
The following is a summary of the changes made in this revision.

- Updated the programming file size details for MPF300 devices. For more information, see Table 2, page 10 and Table 3, page 10.
- Updated the port details of SPI block in Figure 34, page 42 and Figure 37, page 44.

1.4 Revision 3.0
The following is a summary of the changes made in this revision.

- Updated the document for Libero SoC PolarFire v2.0.
- Updated the programming file size details for MPF300 devices. For more information, see Programming File Size, page 10.
- Updated the information about I/O states for XCVR reference clock inputs, XCVR data I/O, GPIO, and HSIO. For more information, see Table 22, page 37.
- Added the procedure to program an external SPI flash. For more information, see Programming External SPI Flash Using Libero, page 42.
- Updated JTAG pin details. For more information, see Table 4, page 12.
- Updated the diagram showing the recommended configuration for SPI master programming. For more information, see Figure 17, page 20.
- Replaced VDDIx with VDDI3 throughout the document.

1.5 Revision 2.0
The following is a summary of the changes made in revision 2.0 of this document.

- Updated the information about bitstream generation. For more information, see Bitstream Generation, page 4.
- Updated the information about auto update. For more information see Auto Update, page 25.

1.6 Revision 1.0
The first publication of this document.
PolarFire® FPGAs offer a variety of programming options to cater to diverse end-user applications. The following components of PolarFire devices are programmable:

- FPGA fabric
- Secure non-volatile memory (sNVM)
- User security settings (keys, passcodes, and locks)

The device can be programmed using on-chip system controller through its dedicated JTAG or SPI interface. Based on the interface used, three programming modes are supported: JTAG, SPI master, and SPI slave.

In JTAG and SPI slave programming modes, the device can be programmed either using an external master such as a microprocessor or a Microsemi FlashPro programmer (version 5 or later). The external master fetches the programming data (bitstream) from an external memory.

In SPI master programming mode, the system controller acts as the master and fetches the bitstream from an external SPI flash memory to program the device. This mode supports two programming features: auto update and in-application programming (IAP). In auto update, the device reprograms itself on power-up, and in IAP, the device is programmed when the user application initiates programming.

*Figure 1* • PolarFire FPGA Programming Modes

- JTAG
  - Using FlashPro Programmer
  - Using External Microprocessor
- SPI Master
  - Auto Update
  - IAP
- SPI Slave
  - Using FlashPro Programmer
  - Using External Microprocessor
The following block diagram shows the PolarFire FPGA programming modes and the associated interfaces.

**Figure 2 • PolarFire Device Programming Modes and Interfaces**

**Note:** When PolarFire FPGAs are used in System controller suspend mode, device programming is disabled to protect the device from unintended programming because of single event upsets. After device initialization, the system controller is held in reset state and cannot provide system services such as security, IAP, or auto update programming. After the device exits System controller suspend mode, it can be programmed as usual.
2.1 Bitstream Generation

The Libero® SoC PolarFire design suite generates the programming bitstream required for various programming modes. Depending on the requirement, the programming bitstream may contain one or more of the following components: the FPGA fabric logic, the sNVM data, and user security settings.

The following table lists the programming interfaces used in various programming modes and the associated bitstream formats.

<table>
<thead>
<tr>
<th>Programming Mode</th>
<th>Interface</th>
<th>Master</th>
<th>Bitstream Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>JTAG programming</td>
<td>System controller’s dedicated JTAG</td>
<td>FlashPro programmer</td>
<td>STP</td>
</tr>
<tr>
<td>JTAG programming</td>
<td>System controller’s dedicated JTAG</td>
<td>External microprocessor</td>
<td>DAT</td>
</tr>
<tr>
<td>SPI slave programming</td>
<td>System controller’s dedicated SPI</td>
<td>FlashPro programmer</td>
<td>DAT</td>
</tr>
<tr>
<td>SPI slave programming</td>
<td>System controller’s dedicated SPI</td>
<td>External microprocessor</td>
<td>DAT</td>
</tr>
<tr>
<td>SPI master programming</td>
<td>System controller’s dedicated SPI</td>
<td>System controller</td>
<td>SPI</td>
</tr>
</tbody>
</table>

2.1.1 Bitstream Generation Flow

The following figure shows where bitstream generation occurs in the Libero SoC PolarFire design flow.

Figure 3 • Bitstream Generation in Libero Design Flow

Create Design

Implement Design
1. Synthesis
2. Place and route
3. Verify timing

Constraint Manager
- Pre-synthesis constraints
- Place and route constraints
- Timing constraints

Program and Debug Design
- Configure design initialization data and memories
- Configure device I/O states during programming
- Configure programming options and security
- **Generate bitstream**
- Program the device

Handoff Design for Production
Export bitstream (.STP, .DAT, .SPI, and .SVF)
2.1.2 Adding sNVM Data to the Bitstream

The sNVM is a user non-volatile flash memory that can be programmed independently. Each PolarFire FPGA has 56 Kbytes of sNVM.

To add multiple sNVM data clients to the bitstream, in Libero SoC PolarFire, go to Design Flow > Program and Debug Design > Configure Design Initialization Data and Memories, as shown in the following figure.

Figure 4 • Design and Memory Initialization

2.1.3 Adding User Security Settings to the Bitstream

PolarFire devices are provisioned with a set of unique factory keys. In addition, the end users can also enroll their own security keys, thus providing complete independence from using Microsemi provided keys. The user encryption key1 (UEK1) and user encryption key2 (UEK2) are user-defined AES-2 symmetric keys. Either of these keys can be used as the root key for encrypting and decrypting bitstreams, and to authenticate them.
To add user security settings in the bitstream:

1. In Libero SoC PolarFire, go to **Design Flow > Program and Debug Design > Configure Security > Custom security options**, as shown in the following figure.

*Figure 5*  Configure Security—Custom Security Options
2. Click **Next** to modify Update policy.

The Configure Security wizard appears, as shown in the following figure.

*Figure 6 • Configure Security Wizard—Update Policy*

If **Back Level protection** is enabled, the **Back Level version** must be lower than the version of the design being programmed. For more information about the fields, click **Help**. The back-level version value restricts the design version that the device accepts as an update. Only (new) programming bitstreams with a Design Version strictly greater than the current Back Level Version previously stored in the device are allowed for programming. Back-level protection is secured by FlashLock/UPK1, which can be bypassed. The back level version and design version can be modified in the configure programming options tool. For more information on sNVM and security settings, see UG0753: PolarFire FPGA Security User Guide.
The following figure shows the configuration of programming options.

**Figure 7 • Configure Programming Options**

For more information on the bypass back level protection, see Bypass Back Level Protection Use Case, page 37.
2.1.4 Configuring Bitstream Components

To configure security settings, fabric, and sNVM of the bitstream, follow these steps:

1. In Libero SoC PolarFire, go to Design Flow > Program and Debug Design > Program Design.
2. Right-click Generate Bitstream, and select Configure Options.... The Configure Bitstream window opens.
3. Select Custom security, Fabric, and sNVM.
4. Click OK.

Figure 8 • Configure Bitstream Window

To export bitstream files, go to Design Flow > Handoff Design for Production > Export Bitstream.

Note: Use Custom Security only bitstream on a blank device.
2.1.5 **Programming File Size**

Programming files are encrypted with factory key or user key. So the file (.dat or .spi) can not be compressed to reduce the file size. The following table lists the programming file sizes when custom security is disabled.

<table>
<thead>
<tr>
<th>Fabric and sNVM (kB)</th>
<th>STAPL</th>
<th>DAT</th>
<th>SPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPF100</td>
<td>5585</td>
<td>3497</td>
<td>3496</td>
</tr>
<tr>
<td>MPF200</td>
<td>9611</td>
<td>6043</td>
<td>6041</td>
</tr>
<tr>
<td>MPF300</td>
<td>14772</td>
<td>9307</td>
<td>9305</td>
</tr>
<tr>
<td>MPF500</td>
<td>23446</td>
<td>14789</td>
<td>14788</td>
</tr>
</tbody>
</table>

The following table lists the programming file sizes when custom security is enabled.

<table>
<thead>
<tr>
<th>Custom Security, Fabric and sNVM (kB)</th>
<th>STAPL</th>
<th>DAT</th>
<th>SPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPF100</td>
<td>5595</td>
<td>3502</td>
<td>3498</td>
</tr>
<tr>
<td>MPF200</td>
<td>9621</td>
<td>6047</td>
<td>6044</td>
</tr>
<tr>
<td>MPF300</td>
<td>14784</td>
<td>9312</td>
<td>9308</td>
</tr>
<tr>
<td>MPF500</td>
<td>23456</td>
<td>14794</td>
<td>14790</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Custom Security (kB)</th>
<th>STAPL</th>
<th>DAT</th>
<th>SPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPF100</td>
<td>71</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>MPF200</td>
<td>71</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>MPF300</td>
<td>72</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>MPF500</td>
<td>71</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Custom Security (kB)</th>
<th>STAPL</th>
<th>DAT</th>
<th>SPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPF100</td>
<td>NA</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>MPF200</td>
<td>NA</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>MPF300</td>
<td>NA</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>MPF500</td>
<td>NA</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
2.2 **Device Programming Flow**

The device programming flow starts when the system controller receives or initiates device programming and ends when the bitstream data is fully transferred and verified. The system controller fetches the bitstream data block by block to program the device. Authentication of the bitstream and verification of the programmed contents are part of the programming flow. The security settings are enabled either after erasing the device contents or on completion of device programming. On successful completion of programming, the system controller resets the device to run the programmed design. This programming flow is common to all the programming modes.

The following figure summarizes the PolarFire device programming flow.

*Figure 9 • PolarFire FPGA Programming Flow*
2.2.1 Programming Time

Programming time is the time taken to erase the existing contents of the device, process bitstream data, program the device, and verify the programmed contents. The programmed content is verified as the next block of data is loaded for programming. The simultaneous programming and verification mechanism considerably reduces the total programming time.

The total programming time of PolarFire devices is less than 60 seconds. For information about programming time for specific devices and programming modes, see DS0141: PolarFire FPGA Datasheet.

2.3 Programming Modes

This section describes the three PolarFire FPGA programming modes in detail.

2.3.1 JTAG Programming

PolarFire devices have a built-in JTAG controller that is compliant with the IEEE 1149.1 and IEEE 1532 standards. The JTAG controller communicates with the system controller using a command register that sends the JTAG instruction to be executed and a 128-bit data buffer that transfers any associated data.

2.3.1.1 JTAG Programming Interface

In PolarFire devices, the JTAG pins are located in a dedicated I/O bank 3 VDDI. For information about the I/O states during JTAG programming, see PolarFire FPGA I/O States During Programming, page 37.

The JTAG bank voltages can be set to operate at 1.8 V, 2.5 V, or 3.3 V. The following table lists the JTAG pins.

<table>
<thead>
<tr>
<th>Pin Names</th>
<th>Direction</th>
<th>Weak Pull-Up/Unused Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMS</td>
<td>Input</td>
<td>Yes/DNC</td>
<td>JTAG test mode select.</td>
</tr>
<tr>
<td>TRSTB</td>
<td>Input</td>
<td>Yes¹</td>
<td>JTAG test reset. Must be held low during device operation.</td>
</tr>
<tr>
<td>TDI</td>
<td>Input</td>
<td>Yes/DNC</td>
<td>JTAG test data in.</td>
</tr>
<tr>
<td>TCK</td>
<td>Input</td>
<td>No²</td>
<td>JTAG test clock</td>
</tr>
<tr>
<td>TDO</td>
<td>Output</td>
<td>No/DNC</td>
<td>JTAG test data out.</td>
</tr>
</tbody>
</table>

1. In unused condition, must be connected to VDDI3 through 1 kΩ resistor per pin and must not be shared with any other pins.

2. In unused condition, must be connected to VSS through 10 kΩ resistor.
2.3.1.2 JTAG Timing

Proper operation of JTAG programming depends on the timing relationship between JTAG pins as shown in the following figure. For recommended timing values, see information related to JTAG switching characteristics in DS0141: PolarFire FPGA Datasheet.

*Figure 10 • JTAG Signals Timing Diagram*

2.3.1.3 JTAG Programming Using FlashPro Programmer

Microsemi FlashPro programmer (version 5 or later) can be used to program PolarFire devices through the dedicated JTAG interface. This can be done either using the Libero SoC PolarFire or a standalone FlashPro Express.

The FlashPro programmer connects to the device via a 10-pin programming header using a FlashPro cable (10-pin ribbon), as shown in the following figure.

*Figure 11 • JTAG Programming Using External Programmer*
The following table lists the FlashPro5 header signals.

### Table 5 • FlashPro Header Signals

<table>
<thead>
<tr>
<th>Pin Number</th>
<th>Signal</th>
<th>Direction to FlashPro Programmer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TCK/SCK</td>
<td>Output</td>
<td>JTAG/SPI clock.</td>
</tr>
<tr>
<td>2</td>
<td>GND</td>
<td>Signal reference. GND pins must be connected.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TDO/MISO</td>
<td>Input</td>
<td>JTAG/SPI data output from programming device.</td>
</tr>
<tr>
<td>4</td>
<td>PROG_MODE</td>
<td>Not connected</td>
<td>Unused</td>
</tr>
<tr>
<td>5</td>
<td>TMS/SS</td>
<td>Output</td>
<td>JTAG test-mode select/SPI slave select.</td>
</tr>
<tr>
<td>6</td>
<td>VJTAG/VSPI</td>
<td>Target interface voltage input.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>VPUMP</td>
<td>Not connected</td>
<td>Unused</td>
</tr>
<tr>
<td>8</td>
<td>TRSTB</td>
<td>Output</td>
<td>JTAG test reset.</td>
</tr>
<tr>
<td>9</td>
<td>TDI/MOSI</td>
<td>Output</td>
<td>JTAG/SPI data input to programming device.</td>
</tr>
<tr>
<td>10</td>
<td>GND</td>
<td>GND</td>
<td></td>
</tr>
</tbody>
</table>

A single FlashPro programmer can program multiple Microsemi FPGAs from the same family or from different families in a single JTAG chain. The TDO pin of the JTAG header represents the beginning of the chain. The TDI pin of the last device connects back to the JTAG header, thus completing the JTAG chain. The following types of FPGAs can be added to a JTAG chain:

- Microsemi devices targeted for programming
- Microsemi bypass devices not targeted for programming
- Non-Microsemi bypass devices

When a device is in bypass mode, the device’s data register length is automatically set to 1 and the device stops responding to any programming instructions. To place a device in bypass mode, the instruction register (IR) length must be known. For Microsemi FPGAs, the IR length is obtained automatically by the FlashPro Express. For non-Microsemi FPGAs, the boundary scan description language (BSDL) file, which contains a sequence of boundary scan commands and data, must be loaded, or the IR length must be manually entered in the FlashPro Express. For more information about JTAG chain programming, see the FlashPro User’s Guide.

### Figure 12 • Device Programming in JTAG Chain

For information about power supply requirement and filtering capacitors, see UG0726: PolarFire FPGA Board Design User Guide.
The following figure shows the connections between the programming header and the device.

Figure 13 • Connecting FlashPro Programmer to a PolarFire FPGA
2.3.1.4 JTAG Programming Using External Microprocessor

An external microprocessor can be used to program PolarFire devices through the dedicated JTAG interface. This type of programming requires that the external microprocessor run DirectC, a Microsemi programming solution for FPGAs, and the microprocessor’s GPIO ports drive the JTAG interface.

The DirectC solution supports programming of the FPGA fabric, sNVM, and user security settings. DirectC is used by adding the necessary APIs and compiling the source code to create a binary executable. The binary executable is downloaded to the external microprocessor along with the programming data file. For more information, see the latest version of the DirectC User Guide available on the Documents tab of the Microsemi DirectC solution webpage.

The following figure shows a sample implementation of PolarFire device programming using an external microprocessor running DirectC.

*Figure 14 • Programming Using External Microprocessor*
2.3.2 SPI Slave Programming

PolarFire devices can be programmed using an external SPI master such as an external microprocessor or a FlashPro programmer through the SPI interface. See Table 7, page 18 for the pin settings that must be used to configure the system controller SPI in slave mode.

The SPI slave or master mode is determined by IO_CFG_INTF SPI pin at device Power-on Reset (POR) and cannot be switched dynamically. A power cycle or DEVRST is required to change the SPI configuration from Slave to Master or vice versa by configuring the IO_CFG_INTF pin as mentioned in Table 6, page 17.

When SPI is in Slave mode, Fabric has no access to SPI and the SPI interface is dedicated to the System Controller.

**Note:** Design initialization from an external SPI flash is not supported when device is in SPI slave programming mode. For information about design initialization, see UG0725: PolarFire FPGA Device Power-Up and Resets User Guide.

2.3.2.1 SPI Slave Programming Interface

In addition to the standard SPI signals, PolarFire devices provide two pins, SPI_EN and IO_CFG_INTF, for configuring the SPI controller.

The following table lists the system controller’s SPI pins and specifies what should be done if a pin is not in use (unused condition).

### Table 6 • System Controller SPI Pins

<table>
<thead>
<tr>
<th>SPI Pin Name</th>
<th>Direction</th>
<th>Description</th>
<th>Unused Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCK</td>
<td>Bidirectional</td>
<td>SPI clock.¹</td>
<td>Connect to VSS through a 10 kΩ resistor</td>
</tr>
<tr>
<td>SS</td>
<td>Bidirectional</td>
<td>SPI slave select.¹</td>
<td>Connect to VSS through a 10 kΩ resistor</td>
</tr>
<tr>
<td>SDI</td>
<td>Input</td>
<td>SDI input.¹</td>
<td>Connect to VDDI3 through a 10 kΩ resistor</td>
</tr>
<tr>
<td>SDO</td>
<td>Output</td>
<td>SDO output.¹</td>
<td>DNC</td>
</tr>
<tr>
<td>SPI_EN</td>
<td>Input</td>
<td>SPI enable. 0: SPI output tristated 1: Enabled Pulled up or down through a resistor or driven dynamically from an external source to enable or tri-state the SPI I/O.</td>
<td>Connect to VSS through a 10 kΩ resistor</td>
</tr>
<tr>
<td>IO_CFG_INTF</td>
<td>Input</td>
<td>SPI I/O configuration. 0: SPI slave interface 1: SPI master interface Pulled up or down through a resistor.</td>
<td>Connect to VSS through a 10 kΩ resistor</td>
</tr>
</tbody>
</table>

¹. Shared between the system controller and the FPGA fabric. When the system controller’s SPI is enabled and configured as master, the system controller hands over the control of the SPI to the fabric (after device power-up).

**Note:** When the SPI_EN pin is disabled (driven low) or when the SS is driven low, the system controller’s SPI outputs are tri-stated.
The SPI_EN and IO_CFG_INTF pins must be configured external to the device. This can be done by using jumpers on the board or by bootstrapping, for example. The following table lists the SPI_EN and IO_CFG_INTF configuration for SPI slave programming.

### Table 7 • System Controller’s SPI Configuration

<table>
<thead>
<tr>
<th>SPI Pins</th>
<th>IO_CFG_INTF</th>
<th>SPI Slave Programming</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_EN</td>
<td>0</td>
<td>x</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0 (SPI slave mode)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1 (SPI master mode)</td>
<td>No</td>
</tr>
</tbody>
</table>

#### 2.3.2.2 SPI Slave Programming Using FlashPro Programmer

Microsemi FlashPro programmer (version 5 or later) can be used to program PolarFire devices through the dedicated SPI. This can be done using either the Libero SoC PolarFire or a standalone FlashPro Express. The FlashPro programmer is connected to the device SPI ports, as shown in the following figure.

The target board must provide power to the VDD, VDD18, VDD25, and VDDI3.

### Figure 15 • SPI Slave Programming Using External Programmer

Device Programming using Libero SoC v12.1 via SPI-Slave instead of JTAG is currently not supported. Support for this use model will be added in a future release.
2.3.2.3 **SPI Slave Programming Using External Microprocessor**

An external microprocessor (such as a host PC or another Microsemi FPGA) can be used to program PolarFire devices through the dedicated SPI port, as shown in the following figure. This type of programming requires that the external microprocessor run the Microsemi SPI-DirectC solution. The external microprocessor can also control the SPI_EN, IO_CNF_INTF and DEVRST_N pins to program the device.

SPI-DirectC supports programming of the FPGA fabric, sNVM, and user security settings. SPI-DirectC is used by adding the necessary APIs and compiling the source code to create a binary executable. The binary executable is downloaded to the external microprocessor along with the programming data file. For more information, see the latest version of the **SPI-DirectC User Guide** available on the Documents tab of the Microsemi DirectC solution webpage. The example project (Direct-C installer) is also available on the Downloads tab.

For information about FlashPro header signals, see Table 5, page 14.

*Figure 16 • SPI Slave Programming Using External Microprocessor*
2.3.3 SPI Master Programming

When the system controller SPI is configured as a master, a PolarFire device can program itself. In SPI master programming, the programming images are stored in the external SPI flash memory using the SPI directory. For more information about the SPI directory and about programming the external SPI flash memory, see Programming the External SPI Flash, page 40.

SPI master programming supports auto update and IAP. In auto update programming, if the version of the update image is found to be different from the currently programmed version, the system controller reads the update image bitstream from the external SPI flash memory and programs the device on power up. In IAP, the user application initiates the device program and the system controller reads the bitstream from the external SPI flash memory to program the device. The auto update and IAP operations are atomic and cannot be interrupted by JTAG or SPI slave commands.

The Auto Update feature is not enabled by default and if required, this needs to be enabled using Libero SoC PolarFire. SPI Master mode also supports Auto Programming and Auto Recovery, see Table 8, page 20. These two features are enabled by default and do not require user configuration.

For information about the I/O states during SPI master programming, see PolarFire FPGA I/O States During Programming, page 37.

The following table lists the initiation sources for the features supported by SPI master programming.

<table>
<thead>
<tr>
<th>Programming Feature</th>
<th>Description</th>
<th>Initiation Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto programming</td>
<td>Programs a blank device</td>
<td>Device reset or power cycle</td>
</tr>
<tr>
<td>Auto update</td>
<td>Updates device contents automatically</td>
<td>Device reset, power cycle, or system service request</td>
</tr>
<tr>
<td>IAP</td>
<td>Updates device contents upon user request</td>
<td>System service request</td>
</tr>
<tr>
<td>Auto recovery</td>
<td>Automatically recovers the device from programming failure</td>
<td>Device power failure during programming</td>
</tr>
</tbody>
</table>

The following figure shows the recommended board configuration for SPI master programming. The VDDI3 must match the voltage specified in the datasheet associated with the external SPI flash.

Figure 17 • Recommended Board Configuration for SPI Master Programming
2.3.3.1 **SPI Master Programming Interface**

The SPI_EN and IO_CFG_INTF pins must be configured external to the device. This can be done by using jumpers on the board or by bootstrapping, for example. The following table provides the SPI_EN and IO_CFG_INTF pin configuration details for SPI master programming.

<table>
<thead>
<tr>
<th>SPI Pins</th>
<th>SPI Master Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_EN</td>
<td>IO_CFG_INTF</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>0 (SPI slave mode)</td>
</tr>
<tr>
<td>1</td>
<td>1 (SPI master mode)</td>
</tr>
</tbody>
</table>

**Table 9 • System Controller’s SPI Configuration**

2.3.3.1.1 **System Controller SPI Mode and Clock**

The system controller SPI operates in data transfer mode 3 (SPI mode 3) for SPI flash read operations. Both the clock polarity (SPO/CPOL) and clock phase (SPH/CPHA) for this data transfer mode must be set to high. The system controller’s SPI operates at a fixed clock of 20 MHz.

2.3.3.2 **Core System Services IP**

In SPI master programming, IAP or auto update can be initiated using system services. System services are system controller actions initiated by the fabric user logic through the system controller’s system service interface (SSI). The fabric user logic requires the core systems services IP (CoreSysServices_PF) available in the Libero catalog for initiating the system services.

Writing a 16-bit system service descriptor to the SSI by the fabric user logic triggers a service request interrupt to the system controller. The lower seven bits of the descriptor specify the service to be performed. The upper nine bits specify the address offset (0–511) in the 2-KB mailbox RAM. The mailbox address specifies the service-specific data structure used for any additional inputs to or outputs from the service. The fabric user logic must write additional parameters to the mailbox before requesting a system service. The following table lists the system service descriptor bits.

<table>
<thead>
<tr>
<th>Descriptor Bit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:7</td>
<td>MBOXADDR[10:2]</td>
</tr>
<tr>
<td>6:0</td>
<td>SERVICEID</td>
</tr>
</tbody>
</table>

**Table 10 • System Service Descriptor**

SSI consists of an asynchronous command-response interface that transfers a system service command from the fabric master to the system controller and the status from the system controller back to the fabric master. SSI operates in conjunction with the synchronous mailbox interface that transfers data related to the system service between the fabric master and the system controller. The following figure shows how to interface core system services IP with the fabric user logic.
The fabric master is connected to the core system services IP using the APB interface. The core system services IP can be configured using the Core System Services configurator in Libero SoC PolarFire, as shown in the following figure. For more information, see the *CoreSysServices_PF Handbook.*
The user application can initiate the following system services using the core system services IP:

- Bitstream authentication
- IAP image authentication
- Auto update
- IAP

**Figure 19 • Core System Services Configurator**

### 2.3.3.3 Bitstream and IAP Image Authentication System Services

For security and reliability reasons, the programming bitstream must be authenticated and validated before the device is programmed. Successful authentication of the bitstream prevents auto recovery. While the authentication is in progress, the fabric user logic continues to operate normally, though without access to SPI flash and system services. Before the device is programmed using auto update or IAP, the user application can run the authentication system service.

**Note:** If the bitstream authentication system service is initiated while a new bitstream is being loaded through the JTAG interface, the system service takes precedence, and the JTAG operation fails.
2.3.3.3.1 **Bitstream Authentication System Service**

The bitstream authentication system service parses a bitstream image stored in the SPI flash and verifies the integrity of the bitstream. The following table lists the fields contained in a bitstream authentication service request.

*Table 11 • Bitstream Authentication Service Request*

<table>
<thead>
<tr>
<th>System Service Descriptor Bit Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:7</td>
<td>MBOXADDR[10:2]</td>
<td>Mailbox address. For the format, see Table 12, page 24.</td>
</tr>
<tr>
<td>6:0</td>
<td>23H</td>
<td>Bitstream authentication command code.</td>
</tr>
</tbody>
</table>

The following table describes the bitstream authentication service mailbox format.

*Table 12 • Bitstream Authentication Service Mailbox Format*

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Parameter</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>SPIADDR</td>
<td>Input</td>
<td>Address of the bitstream in SPI flash. If the external SPI flash device does not support 32-bit addresses, SPIADDR[31:24] is ignored.</td>
</tr>
</tbody>
</table>

2.3.3.3.2 **IAP Image Authentication System Service**

The IAP image authentication system service parses an image stored in the SPI flash and verifies the integrity of the image descriptor, bitstream, and design initialization data.

The following table lists the fields contained in an IAP image authentication service request.

*Table 13 • IAP Image Authentication Service Request*

<table>
<thead>
<tr>
<th>System Service Descriptor Bit Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Reserved.</td>
<td></td>
</tr>
<tr>
<td>14:7</td>
<td>IMAGEID[7:0]</td>
<td>Identifies the image index in the SPI directory for image authentication.</td>
</tr>
<tr>
<td>6:0</td>
<td>22H</td>
<td>Authenticates image command.</td>
</tr>
</tbody>
</table>

2.3.3.3.3 **Authentication Service Status Codes**

If bitstream authentication or IAP image authentication is successful, the status code 0 is generated. If bitstream authentication or IAP image authentication fails, an 8-bit error code is generated. For detailed information about the error codes, see Appendix: Error Codes, page 37.

2.3.3.3.4 **Usage of Authentication System Services**

The programming image contains the image descriptor, bitstream, and optional design initialization data. The bitstream authentication system service can be used to authenticate the bitstream only. The IAP image authentication system service, however, can be used to authenticate the entire programming image, including the image descriptor, bitstream, and optional design initialization data.
2.3.3.4 Auto Update

For auto update to occur, the auto update feature needs to be enabled in the user design. On power-up, the device selects the newer version of the first two images stored in the SPI directory. If the version of the newer image does not match that of the currently programmed image, then auto update occurs. The following figure shows the high-level flow of auto update programming.

*Figure 20 • Auto Update High-Level Flowchart*

*Different scenarios to reach here:*
- Device is blank and auto update is initiated to program the device
- As part of IAP recovery when power fails during IAP or partially programmed with an invalid image
- As part of auto update recovery when power fails during auto update
The following figure shows the detailed flow of auto update programming.

**Figure 21 • Auto Update Detailed Flow**

1. Read image descriptor pointers.
2. Read Image 0 and Image 1 version info and determine update image.

- Design update required?1
  - Yes
    - Design not updated
  - No
    - Program newer image
      - Program Passed?
        - Yes
          - Image info authenticated?2
            - Yes
              - Retry program newer image
            - No
              - Program passed?
                - No
                  - Program older image
                - Yes
                  - Program passed?
                    - Yes
                      - Initialize design
                    - No
                      - Design not updated

1. Condition for update: version of the design differs from the update image or the device is blank.
2. Device checks only BITS and AUTH components of the bitstream as part of the programming.
3. The device is not programmed, and user intervention is required.
The following table lists example auto update conditions when different image versions are available in the SPI flash.

### Table 14 • Example Auto Update Conditions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank device</td>
<td>2, 3</td>
<td>Disabled</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2, 3</td>
<td>Disabled</td>
<td>No auto update</td>
</tr>
<tr>
<td>2</td>
<td>1, 2</td>
<td>Disabled</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1, 2</td>
<td>Disabled</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3, 4</td>
<td>Enabled and set to 4</td>
<td>No auto update</td>
</tr>
<tr>
<td>3</td>
<td>3, 5</td>
<td>Enabled and set to 4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3, 5</td>
<td>Enabled and set to 4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>2, 3</td>
<td>Enabled and set to 4</td>
<td>No auto update</td>
</tr>
</tbody>
</table>

#### 2.3.3.4.1 Auto Update on a Blank Device (Auto Programming)

When a blank device is powered up or reset (with SPI master mode enabled), the device programs itself using the newest version of the image. This process is known as auto programming.

#### 2.3.3.4.2 Auto Update on a Preprogrammed Device

Auto update is also initiated through system services on a preprogrammed device. If the device is preprogrammed, it compares the update image with the currently programmed image. If the version of the update image is found to be different from the currently programmed version, auto update programming is initiated.

To perform auto update on a preprogrammed device, the user application must initiate a system service request using the core system services IP. The system controller executes the system service request and programs the device.

The user application cannot obtain the status code in the following scenarios:

- If the auto update program is successful, the device is automatically restarted to initialize the new version of the design.
- If the auto update program fails, the auto update recovery procedure attempts to program the device with the valid image again.

The following table lists the fields in an auto update system service request.

### Table 15 • Auto Update System Service Request

<table>
<thead>
<tr>
<th>System Service Descriptor Bit Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:7</td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>6:0</td>
<td>46H</td>
<td>Auto update programming command.</td>
</tr>
</tbody>
</table>

When auto update is not enabled in the user design, the auto update system service can be used to update the device with the newest image using the core system services IP.

#### 2.3.3.4.3 Recovery on Auto Update Programming Failure

When power fails during auto update programming, the auto update programming flow is initiated on the next boot cycle to program the device with the newest image.
**Note:** If the device fails to program the newer image, it retries once before programming itself with the older version of the image. If the device remains blank at the end of auto update, there is no indication through I/O and user intervention is required.

### 2.3.3.4 Enabling Auto Update Option in User Design

To enable auto update, follow these steps:

1. Click **Configure Design Initialization Data and Memories**, and select the **SPI Flash** tab.
2. Select the **Enable Auto Update** checkbox.

**Figure 22 • Auto Update Setting**

3. Click **Configure Programming Options**, and specify the design version and back level version, as shown in the following figure.

**Figure 23 • Design Version**
2.3.3.4.5 Auto Update Use Models

Auto update is initiated when a different version of the programming image is available in the SPI flash memory. For more information, see SPI Directory, page 40. The device uses the Bits/Version component of the programming image to determine the version. The Bits/Version component appears at the beginning of a bitstream and contains version information. This section describes three auto update use models—ping pong, golden image, and single image. Based on the design requirement, any of these models can be used.

Ping Pong

Auto update uses the newer of the first two images on the SPI flash memory. When a new image is written to the SPI flash memory, the older of the two images is overwritten with the new image. This is known as the ping pong model and is used when the previous image version needs to be retained along with the newer image. This facilitates an automatic rollback to the previous image if the new image fails. The following illustration shows the ping pong use model.

Figure 24 • Ping Pong Use Model
Golden Image

When auto update fails with a newer version of the image, the device needs to be updated safely using a working image. This image is known as the golden image. When a new image is written to the SPI flash memory, it must not overwrite the golden image. The following figure shows the golden image use model.

Figure 25 • Golden Image Use Model
Single Image
This model is used when only one image is available for updating the device. The following figure shows the single image use model.

Figure 26 • Single Image Use Model
2.3.3.5 IAP

IAP reprograms the device with a specific programming image. In IAP, regardless of the image version, the device chooses the programming image based on either the image index or the SPI image address. The fabric user logic specifies the programming image and initiates reprogramming of the device using the IAP system service.

2.3.3.5.1 IAP Using System Service

The user application initiates an IAP system service request using the core system services IP. The system service specifies whether the image is used for verification or programming. The system controller automatically reads the bitstream from the SPI flash to verify or program the device contents.

Verify Operation

The verify operation compares the specified programming image contents with the device contents. The following table lists the fields in an IAP system service request using the image index.

<table>
<thead>
<tr>
<th>System Service Descriptor Bit Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>14:7</td>
<td>SPI_IDX[7:0]</td>
<td>Identifies the image index in the SPI directory for IAP operation.</td>
</tr>
<tr>
<td>6:0</td>
<td>44H</td>
<td>IAP verify operation.</td>
</tr>
</tbody>
</table>

An SPI flash memory address can be specified instead of the image index within the SPI directory, as shown in the following table.

<table>
<thead>
<tr>
<th>System Service Descriptor Bit Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:7</td>
<td>MBOXADDR[10:2]</td>
<td>Mailbox address. For the format, see Table 20, page 33.</td>
</tr>
<tr>
<td>6:0</td>
<td>45H</td>
<td>IAP verify operation.</td>
</tr>
</tbody>
</table>

Table 16 • IAP Verify Request by Image Index

If the IAP verification is successful, the status code 0 is generated. If the IP verification fails, an 8-bit error code is generated. For detailed information about the error codes, see Appendix: Error Codes, page 45.

Note: Digest check system service is recommended to verify the integrity of the device contents instead of IAP verify operation. For more information, see DS0141: PolarFire FPGA Datasheet.

Program Operation

The program operation updates the device contents using a specified programming image. The IAP program operation does not authenticate the image before executing the program. The image can be authenticated using the IAP image authentication system service. For more information, see IAP Image Authentication System Service, page 24.

The user application cannot obtain the status code in the following scenarios:

- If IAP is successful, the device is automatically restarted to initialize the new design.
- If IAP fails, the IAP recovery procedure attempts to program the device with image 0.

Note: IAP recovery considers image 0 when the pointer to image 1 in the SPI directory is null. For more information, see SPI Directory, page 40.
The following table lists the fields in an IAP system service request using the image index.

Table 18 • IAP Program Request by Image Index

<table>
<thead>
<tr>
<th>System Service Descriptor Bit Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td>Reserved.</td>
</tr>
<tr>
<td>14:7</td>
<td>SPI_IDX[7:0]</td>
<td>Identifies the image index in the SPI directory for IAP operation.</td>
</tr>
<tr>
<td>6:0</td>
<td>42H</td>
<td>IAP program operation.</td>
</tr>
</tbody>
</table>

An SPI flash memory address can be specified instead of the image index within the SPI directory, as specified in the following table.

Table 19 • IAP Request by Image Address

<table>
<thead>
<tr>
<th>System Service Descriptor Bit Field</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:7</td>
<td>MBOXADDR[10:2]</td>
<td>For the mailbox format, see the following table.</td>
</tr>
<tr>
<td>6:0</td>
<td>43H</td>
<td>IAP program operation.</td>
</tr>
</tbody>
</table>

The following table describes the mailbox format.

Table 20 • Mailbox Format

<table>
<thead>
<tr>
<th>Offset</th>
<th>Length (bytes)</th>
<th>Parameter</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>SPIADDR</td>
<td>Input</td>
<td>Programming image address in SPI flash memory. If the attached SPI flash device does not support 32-bit addresses, SPIADDR[31:24] is ignored.</td>
</tr>
</tbody>
</table>

2.3.3.5.2 Recovery on Programming Failure

When power fails during IAP, the device programs itself with image 0.

Note: When the device fails to program the specific image, it retries once before programming itself with image 0. If the device is still blank at the end of IAP, there is no indication through I/O and user intervention is required.
2.3.3.5.3 IAP Flow

The following illustration shows the IAP flow.

**Figure 27 • IAP Flowchart**

1. Device checks only BITS(starting bits of the bitstream) and AUTH(encryption keys information) components of the bitstream as part of the programming.
2.3.3.5.4 IAP Use Model

PolarFire devices support the multi-image IAP use model, which allows up to 255 images to be stored in the SPI flash memory. The image descriptor pointers are located in Sector 0 of the SPI flash memory. The device can be programmed with any image; however, if the program fails, the device is programmed with image 0. The programming image pointer next to the image 0 pointer must be null (empty slot). This model is used when the device needs to be updated with a specific image from among the available images. The following illustration shows the multi-image use model.

Figure 28 • Multi-Image Use Model
2.4 Bypassing the Back Level Protection

If Back Level protection is enabled in Configure Security tool, the back level protection can be bypassed for SPI bitstreams while exporting the bitstream using libero. To prevent Programming Recovery failures enable the Bypass the Back Level Protection for Recovery/Golden bitstream (SPI files only) as shown in following figure.

Figure 29 • Selecting Bypass Back Level Protection Feature

When the SPI bitstream is added to SPI flash using design and memory initialization data, the tool shows back level protection bypass feature in bitstream as shown in following figure.

Figure 30 • Status of Bypass Back Level Protection
2.4.1 Bypass Back Level Protection Use Case

The following table describes the user case for Bypass Back Level Protection.

**Table 21 • Bypass Back Level Protection Use Case**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Golden/Recovery</td>
<td>Auto Programming</td>
<td>Pass</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>IAP/Update Bitstream</td>
<td>Auto Update/IAP</td>
<td>Pass</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>IAP/Update Bitstream</td>
<td>Auto Update/IAP</td>
<td>Fail, Attempt Programming Recovery</td>
<td>4</td>
<td>Not Enabled</td>
<td>2</td>
</tr>
</tbody>
</table>

The steps are described as follows:

1. The device programs with a bitstream version 2 and back level version 1. The current device back level version is set to 1.
2. The device then updates with a bitstream version 3 and back level version 2. The current device back level version is set to 2.
3. The device attempts to update itself with a bitstream version 4 and fails to update. In this case, the device attempts to recover using a golden/recovery bitstream version 2. But the recovery also fails as the current device back level protection is set to version 2 and the golden/recovery bitstream version is equal to the back level version. The **Bypass Back Level Protection must be enabled** (see Figure 29, page 36) for Golden/Recovery bitstream to avoid programming recovery failures because of back level protection.

2.5 PolarFire FPGA I/O States During Programming

The following table lists the I/O states that apply during various stages of programming.

**Table 22 • I/O States for Various Programming Modes**

<table>
<thead>
<tr>
<th>I/O Type</th>
<th>JTAG Programming</th>
<th>SPI Slave Programming</th>
<th>SPI Master Programming (IAP/Auto Update)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>May be kept alive during IAP using loopback mode, allowing the XCVR link to be kept active.</td>
</tr>
</tbody>
</table>
| XCVR data I/O    | As set by the boundary scan cell. | Not affected.         | Not affected.                           
|                  |                   |                       | May be kept alive during IAP using loopback mode, allowing the XCVR link to be kept active. |
| GPIO and HSIO    | I/Os are enabled, but the I/O state can be set using the boundary scan cell. | Can be weakly pulled up using the SPI slave instruction ISC_ENABLE. | Outputs are tri-stated and weakly pulled up. |
In Libero SoC PolarFire, the I/O states can be set before JTAG programming, and these I/O states are held at the set values during JTAG programming. The following are the I/O output state settings:

- 1: I/O is set to drive out logic high
- 0: I/O is set to drive out logic low
- Last Known State: I/O is set to the last value that was driven out before entering the programming mode and then held at that value during programming
- Z: I/O is tri-stated

The I/O output states can be set as shown in the following figure.

**Figure 31 • I/O States During Programming (JTAG Mode Only)**

### 2.6 Programming Recommendations

To ensure successful programming, the following guidelines are recommended to be used:

- Authenticate the bitstream before programming the device.
- Do not assert the reset pin (DEVRST_N) during programming because this may corrupt the device configuration.
- Use the correct configuration and programming interface based on the selected programming mode.
- Configure the device I/O states (before JTAG programming) based on the design requirements. For more information, see PolarFire FPGA I/O States During Programming, page 37.

### 2.7 Brownout During Programming

Brownout is a condition that occurs when the power supplies fall below recommended levels. If brownout occurs during programming, the device automatically recovers from the programming failure (since auto recovery is enabled by default) and programs the device with a valid programming image stored in external SPI flash.
2.8 **Zeroization**

PolarFire devices have a built-in capability that can zeroize (clear and verify) any or all configuration storage elements as per the user setting. Internal volatile memories such as LSRAMs, uSRAMs, System Controller RAMs are cleared and verified. Once the zeroization is complete, a zeroization certificate can be retrieved using a JTAG/SPI slave instruction to confirm that the zeroization process is successful. For more information on zeroization, see *UG0753: User Guide PolarFire FPGA Security*
3 Programming the External SPI Flash

To perform IAP or auto update, an external SPI flash memory is required. This SPI flash memory interfaces with the system controller’s SPI and stores the programming images.

The SPI flash memory is divided into several sectors. The 1KB memory in first sector (sector 0) is used as the SPI directory, and it contains the programming image indexes (descriptor pointers). The remaining flash memory stores the programming images.

3.1 Supported SPI Flash Devices

SPI flash devices from various vendors implement a standard instruction set for read operations. The system controller firmware executes the following command to identify the addressing mode (3-byte or 4-byte):

**READ SERIAL FLASH DISCOVERY PARAMETER (5AH)**

The system controller supports devices from Micron, Winbond, Macronix, and Spansion. However, any other device compatible with the JESD216 standard may also be used. Devices that are not JESD216-compliant may still be used if they support the FAST READ (0BH) command with 3-byte addressing. Such devices are limited to using only the first 128 Mb of the flash memory.

3.2 SPI Directory

The SPI directory is a collection of image descriptor pointers that point to the beginning of the programming image. Each pointer uses four bytes. If the SPI flash memory device supports only the 3-byte addressing mode, the first three bytes are used.

For IAP recovery to choose image 0 on power up, the programming image pointer next to the image 0 pointer must be null (empty slot), otherwise auto update is chosen. The following figure shows the SPI flash directory with the programming image descriptor pointers.

*Figure 32 • SPI Flash Directory*
3.3 Use Models for Programming SPI Flash

The external SPI flash can be programmed either using JTAG or the system controller’s SPI. When the system controller’s SPI is enabled and configured in SPI master mode, the system controller’s SPI port is shared between the system controller and either the FPGA fabric master or JTAG. This section describes the use models for programming the external SPI flash.

3.3.1 Programming the SPI Flash Using External Processor

When the SPI_EN pin is disabled (driven low), the system controller’s SPI outputs are tri-stated, and the external processor can drive the SPI pins to program the SPI flash. Neither the system controller nor the fabric can drive the SPI interface. The external processor can drive the SPI_EN pin low to program the external SPI flash. The SPI_EN pin can also be configured external to the device using the jumpers on the board. The following figure shows the connections required for programming the SPI flash using an external processor.

*Figure 33 • SPI Flash Programming Using External Processor*
3.3.2 Programming the SPI Flash Using JTAG

The external SPI flash can be programmed using a FlashPro programmer (version 5 or later) through the system controller’s JTAG interface. The JTAG controller uses a special JTAG instruction—SPIPROG (IR=0xb0)—to interface with the external SPI flash through the system controller’s SPI. The JTAG controller in PolarFire devices supports this instruction to directly drive the system controller’s SPI outputs. The following figure shows the connections required for programming the SPI flash using JTAG.

![Figure 34 • SPI Flash Programming Using JTAG](image)

3.3.2.1 Programming External SPI Flash Using Libero

The Libero SoC software allows you to program the external SPI flash memory with programming images. To program the SPI flash memory:

1. Go to Design Flow > Program and Debug Design > Configure Design Initialization Data and Memories, and select the SPI Flash tab, as shown in the following figure.

![Figure 35 • SPI Flash Programming in Libero SoC](image)

2. Under SPI Flash Clients, add the required programming images, and click Apply. For more information about values to be entered in the fields, click Help.
3. Double-click **Run PROGRAM_SPI_IMAGE Action** to get the SPI flash programmed with the SPI directory and the programming images.

**Figure 36 • Run PROGRAM_SPI_IMAGE Action**

For more information on design initialization data and memories, see section **How To Set Up Design and Memory Initialization** from **UG0725 User Guide PolarFire FPGA Device Power-Up and Resets**.

**Note:** The following are the recommendations for SPI Flash Programming Using Libero.

- Only the Micron SPI Flash is currently supported with the PolarFire Evaluation/Splash Kit.
- This tool erases the SPI Flash prior to programming. It is recommended to program the SPI Flash with Libero SoC prior to programming other data on the SPI Flash using non-Libero programming solutions.
- Partial update of the SPI Flash is currently not supported.
- It is not recommended to have large gaps between clients in the SPI Flash, since gaps are currently programmed with 1’s and increases programming time.
3.3.3 Programming the SPI Flash Using Fabric User Logic

When the system controller’s SPI is enabled and configured as master, the system controller hands over the control of the SPI to the fabric (after device power-up). The JTAG controller that starts programming the SPI flash, or any system service request from the fabric user logic, can take over the control of SPI from the fabric.

The PolarFire FPGA fabric user logic gets the programming images from an external memory source, as shown in the following figure. The fabric user logic accesses the external SPI flash using the CoreSPI controller and PF_SPI macro provided in Libero Catalog.

**Note:** To fetch the programming images and write to the external SPI flash, PolarFire devices must be preprogrammed with a design. For more information, see AC466: PolarFire FPGA Auto Update and In-Application Programming Application Note.
Appendix: Error Codes

The system controller executes system service requests from the design using the system service interface (SSI). When a service is completed, a status code is written to the SSI. This status code can be 0 (success) or an 8-bit error code. The following table lists the error codes.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Validator or hash chaining mismatch</td>
<td>Bitstream is constructed incorrectly, or a wrong security key is used.</td>
</tr>
<tr>
<td>2</td>
<td>Unexpected data received</td>
<td>Additional data is received after the end of the bitstream (EOB) component.</td>
</tr>
<tr>
<td>3</td>
<td>Invalid/corrupt encryption key</td>
<td>Requested key mode is disabled, or the key could not be read or reconstructed.</td>
</tr>
<tr>
<td>4</td>
<td>Invalid component header</td>
<td>Bitstream contains invalid component data.</td>
</tr>
<tr>
<td>5</td>
<td>Back level not satisfied</td>
<td>Bitstream version is older than that of the current back level in the device.</td>
</tr>
<tr>
<td>6</td>
<td>Illegal bitstream mode</td>
<td>Requested bitstream mode is disabled by user security.</td>
</tr>
<tr>
<td>7</td>
<td>DSN binding mismatch</td>
<td>Bitstream is rejected because the device serial number (DSN) in the bitstream does not match the DSN on the device.</td>
</tr>
<tr>
<td>8</td>
<td>Illegal component sequence</td>
<td>Bitstream ends in the ERR state, meaning it is an illegal bitstream. Every bitstream begins in the BEGIN state, but only a legal bitstream ends in the END state.</td>
</tr>
<tr>
<td>9</td>
<td>Insufficient device capabilities</td>
<td>Bitstream is rejected because the capabilities specified in the bitstream do not match the target device’s capabilities.</td>
</tr>
<tr>
<td>10</td>
<td>Incorrect DEVICEID</td>
<td>Bitstream is rejected because an attempt by the DEVICEID specified in the bitstream does not match the part identification field of the target device.</td>
</tr>
<tr>
<td>11</td>
<td>Unsupported bitstream protocol version (regeneration required)</td>
<td>Bitstream is rejected because of an attempt made by the old device to decode the new version of bitstream or by the new device to decode the old version of the bitstream.</td>
</tr>
<tr>
<td>12</td>
<td>Verify not permitted on this bitstream</td>
<td>When the device programs the bitstream with encryption keys, it is not possible to use the bitstream later to verify the device contents because the device refers to the modified encryption keys.</td>
</tr>
<tr>
<td>13</td>
<td>Invalid device certificate</td>
<td>Device certificate is missing or invalid.</td>
</tr>
<tr>
<td>14</td>
<td>Invalid DIB</td>
<td>Device integrity bits are invalid.</td>
</tr>
<tr>
<td>21</td>
<td>Device not in SPI master mode</td>
<td>Bitstream is executed in IAP mode, but the device is not configured as SPI master.</td>
</tr>
<tr>
<td>22</td>
<td>No valid images found (auto update)</td>
<td>Bitstream is executed through auto update mode, but no valid image pointers are found.</td>
</tr>
<tr>
<td>23</td>
<td>No valid images found (IAP)</td>
<td>Bitstream is executed through IAP via index mode, but no valid image pointers are found.</td>
</tr>
</tbody>
</table>
### Table 23 • Error Codes (continued)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>Programmed design version newer than auto update image</td>
<td>Bitstream is executed through auto update mode, and the design version is the latest.</td>
</tr>
<tr>
<td>25</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Selected image invalid and no recovery performed because the device is running a valid design</td>
<td>Bitstream is executed through auto update or IAP mode, and the selected image is invalid.</td>
</tr>
<tr>
<td>27</td>
<td>Selected recovery image failed to program</td>
<td>Bitstream is executed through auto update or IAP mode, and the selected recovery image failed to program the device.</td>
</tr>
<tr>
<td>127</td>
<td>Abort</td>
<td>A non-bitstream instruction is executed during bitstream loading.</td>
</tr>
<tr>
<td>128</td>
<td>NVMVERIFY</td>
<td>Fabric/security key segment verification failed.</td>
</tr>
<tr>
<td>129</td>
<td>PROTECTED</td>
<td>The device non-volatile memory cannot be modified because of device security settings.</td>
</tr>
<tr>
<td>130</td>
<td>NOTENA</td>
<td>Programming mode is not enabled.</td>
</tr>
<tr>
<td>131</td>
<td>SNVMVERIFY</td>
<td>The sNVM verify operation failed.</td>
</tr>
<tr>
<td>132</td>
<td>SYSTEM</td>
<td>An error occurred in the system hardware (PUF or DRBG).</td>
</tr>
<tr>
<td>133</td>
<td>BADCOMPONENT</td>
<td>An error is detected in a component’s payload.</td>
</tr>
<tr>
<td>134</td>
<td>HVPROGERR</td>
<td>The HV programming subsystem has failed.</td>
</tr>
<tr>
<td>135</td>
<td>HVSTATE</td>
<td>The HV programming subsystem is in an unexpected state because of an error.</td>
</tr>
</tbody>
</table>