SRunner: An Embedded Solution for Serial Configuration Device Programming

June 2008, Version 1.1 Application Note 418

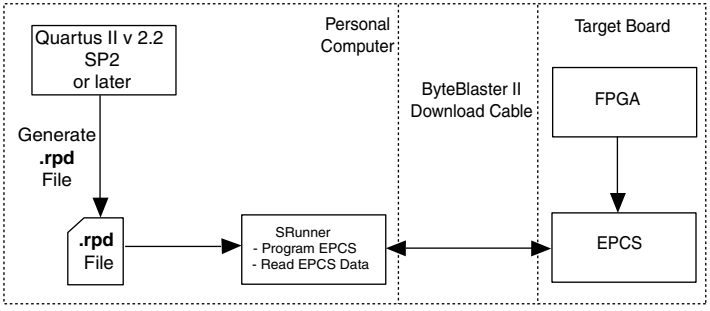
Introduction

The SRunner is a standalone Windows-based software driver that allows users to program Altera® Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) using the ByteBlaster™ II download cable. The SRunner software code is developed and tested on the Windows XP platform.

You can modify and port the SRunner source code to other platforms for EPCS programming.

The input file to the SRunner is a Raw Programming Data (**.rpd**) file that is generated by Quartus® II version 2.2 SP2 and above. The SRunner reads the **.rpd** file and programs the data to the EPCS via the serial interface. Figure 1 shows how the SRunner is conceptualized.

Figure 1. SRunner Concept Overview



Features

The SRunner software driver has the following features:

- Programming the EPCS with a **.rpd** file

  Before programming, the SRunner checks the EPCS silicon ID to determine the maximum size of the EPCS device's memory that ensures the appropriate **.rpd** file size. In addition, all the EPCS data is erased using the `Erase Bulk` operation code before the programming stage.

During programming, the SRunner reads the configuration data from the **.rpd** file and sends the data to the EPCS via the parallel port. The configuration data is programmed into the EPCS using the `Write Bytes` operation code.

- Read back of EPCS data

  With this feature, the SRunner creates a file to store data from the EPCS. The SRunner reads the EPCS data by using the `Read Bytes` operation code and stores the data byte from the EPCS in this file.

- Verify EPCS data

  With this feature, the SRunner verifies the content of the EPCS against the content of data saved in the **.rpd** file. The SRunner reads data by using the `Read Bytes` operation and compare the data stored in the **.rpd** file. If mismatch occurs, SRunner will flag an error to indicate the verification operation has failed.

For more details about the EPCS devices, please refer to *Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) Data Sheet*.

Executing SRunner

You must enter command instructions as shown in Table 1 when executing the SRunner software driver.

Table 1. Instructions for SRunner

Instruction	Action
`srunner -program -<EPCS density> <filename1>.rpd`	To open **<filename1>.rpd** and program the data from the Raw Programming Data file.
`srunner -read -<EPCS density> <filename2>.rpd`	To read out EPCS data and save in the **<filename2>.rpd** file.
`srunner -verify -<EPCS density> <filename1>.rpd`	To verify the EPCS data against the data saved in **<filename1>.rpd** file.

Figure 2 shows the listing in the DOS command window when you program an EPCS device with the `srunner -program -<EPCS density> <filename>.rpd` instruction.

Figure 2. Programming EPCS with the srunner -program -<EPCS density> <filename>.rpd Command

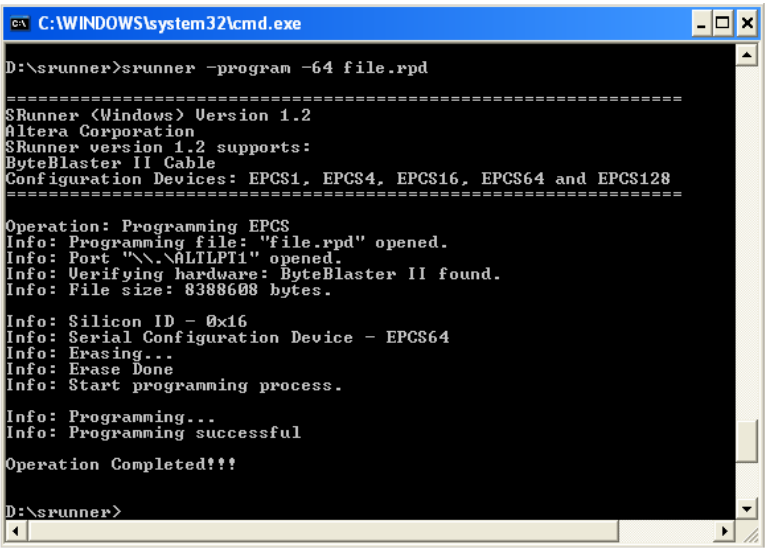


Figure 3 shows the listing in the DOS command window when you read from the EPCS device with the `srunner -read -<EPCS density> <filename>.rpd` instruction.

Figure 3. Reading EPCS Data with the srunner -read -<EPCS density> <filename>.rpd Command

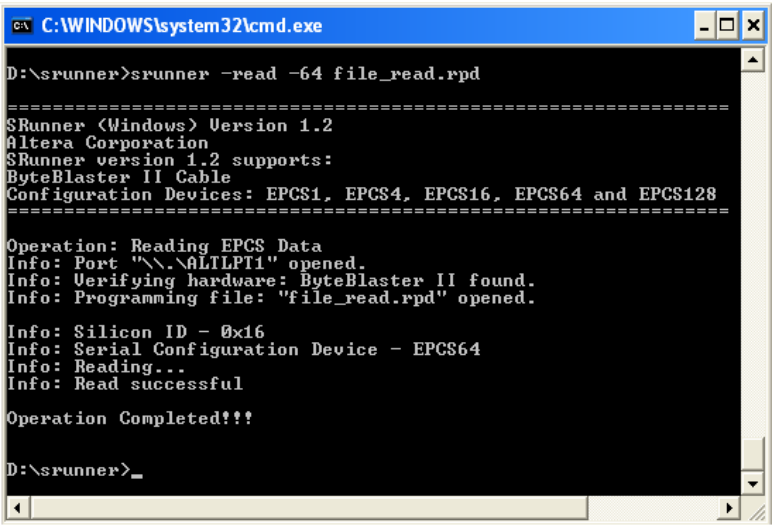


Figure 4 shows how the listing in the DOS command window when you verify the data stored inside the EPCS device with the `srunner -verify -<EPCS density> <filename>.rpd` instruction.

Figure 4. Verifying EPCS Data with the srunner -verify -<EPCS density> <filename>.rpd Command

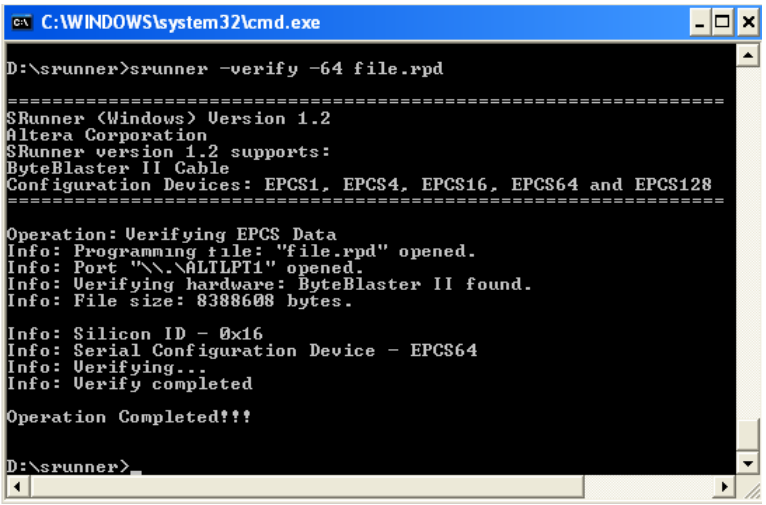
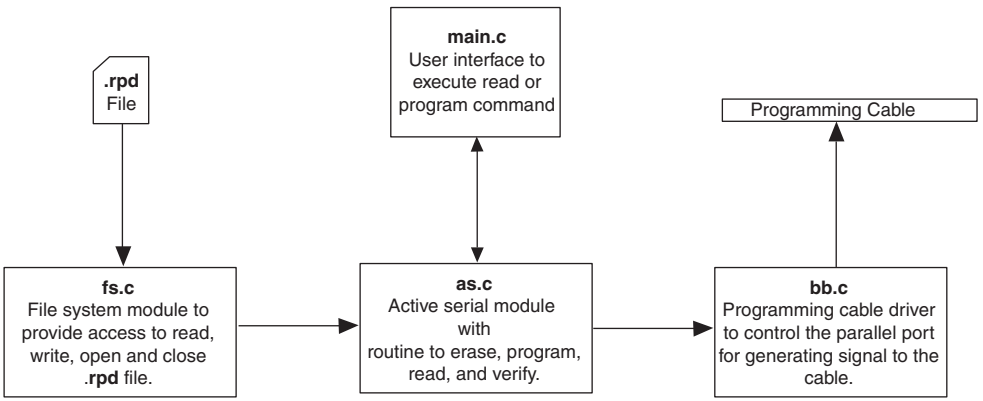
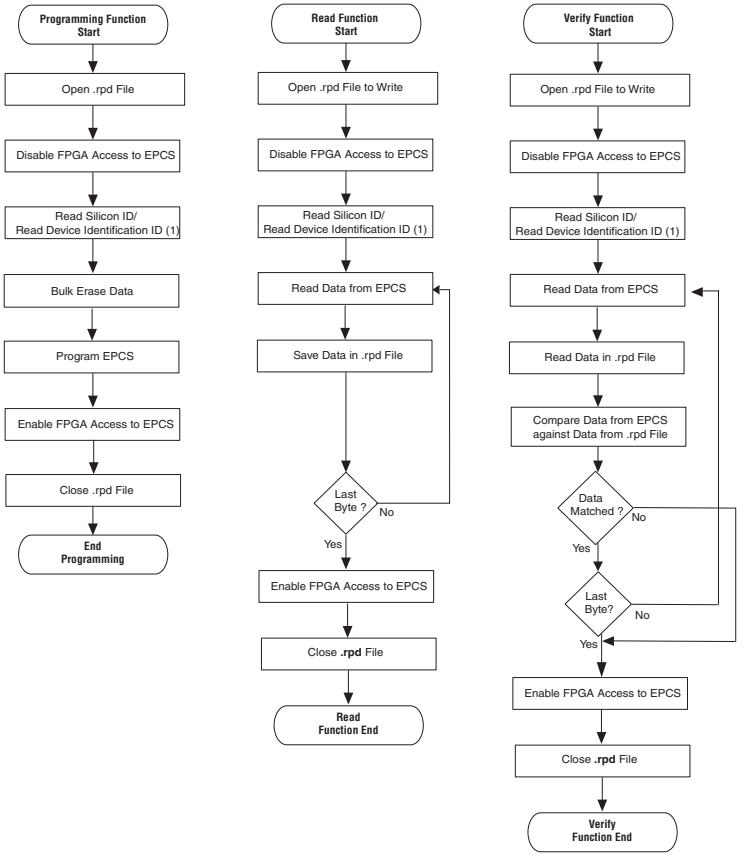


Figure 5. SRunner Software Structure



When you execute a program or a read instruction, the SRunner software driver respectively performs specific program and read steps. Figure 6 shows the program flow during the execution of the program and read instruction using the SRunner.

Figure 6. Program Flow During the Execution of Program, Read and Verify Instructions



Note to Figure 6:

(1) Only EPCS1, EPCS4, EPCS16 and EPCS64 devices support Read Silicon ID operation. The EPCS128 supports Read Device Identification ID operation.

Table 2 lists and describes the source files in SRunner.

Table 2. SRunner Source Files

File	Description
main.c	User interface module that reads user instructions.
as.c as.h	Contains an active serial algorithm to erase, program, read and verify data on the EPCS. The header file contains the Active Serial instruction set.
bb.c bb.h	ByteBlaster II and ByteBlaster MV driver directs the signal to the programming cable (ByteBlaster MV is not used to program EPCS).
fs.c fs.h	File system module to open, close, create and read file (for example, a **.rpd** file).
user.h	User-defined error code.

The active serial module (as.c) source file consists of various function calls that perform the active serial algorithm to erase, program, read and verify data on the EPCS. Table 3 lists and describes the function calls in the active serial module.

Table 3. Functional Description for Active Serial Module (as.c) (Part 1 of 2)

Function	Description
as_program()	Executes during the **srunner -program <filename1>.rpd** command.
as_read()	Executes during **srunner -read <filename1>.rpd** command.
as_open()	Opens the programming file and initialize the parallel port.
as_close()	Closes the programming file and close the parallel port.
as_program_start()	Disables the FPGA access to EPCS by pulling nCE high and nConfig low. Initializes nCS pin to high.
as_program_done()	Enables the FPGA access to EPCS by pulling nCE low and nConfig high. Pulls nCS pin to high.
as_silicon_id()	Checks EPCS Silicon ID to ensure correct device is programmed.
as_bulk_erase()	Erases all the EPCS data.
as_prog()	Checks the Raw Programming Data size to determine the amount of programming data to send to the programming cable. Read data from **.rpd** File and send to the parallel port during EPCS programming.
as_verify()	Reads the EPCS data out and compare with the **.rpd** file to verify the programming process is successful.
as_readback()	Reads back all the EPCS data and store in a **.rpd** file.

Table 3. Functional Description for Active Serial Module (as.c) (Part 2 of 2)

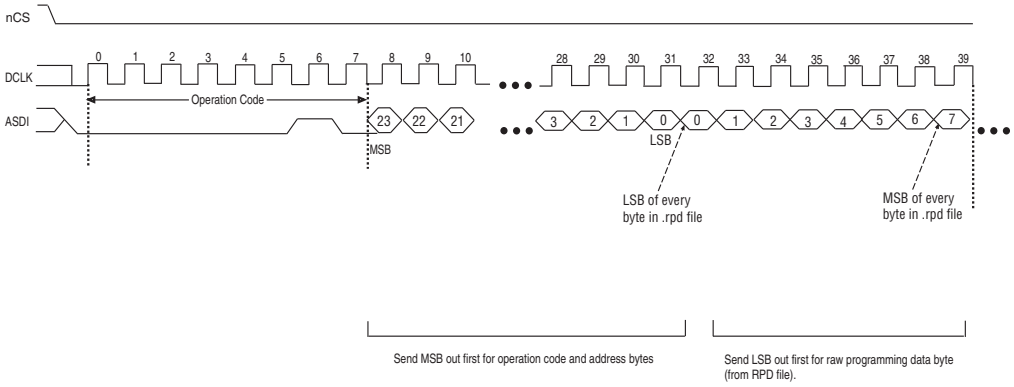
Function	Description
as_program_byte_lsb()	Stores the data byte to the parallel port buffer LSB first and generates a CLK signal (data is sampled by EPCS at rising edge) for programming the data into EPCS.
as_read_byte_lsb()	Reads the EPCS data from the DATAOUT pin and stores it in a byte buffer LSB first and generates a CLK signal (data is read at the rising edge) for reading the EPCS in the Raw Programming Data file format.
as-program_byte_msb()	Stores the data byte to the parallel port buffer MSB first and generates a CLK signal (data is sampled by EPCS at the rising edge) for sending instruction sets and addresses to the EPCS.
as_read_byte_msb()	Reads the EPCS data from the DATAOUT pin, stores it in a byte buffer MSB first and generates a CLK signal (data is read at rising edge) for reading the status register from the EPCS.
as_lsb_to_msb ()	Convert one byte of data by changing the LSB to MSB.

Operation Code and Data Out Sequence Diagram

Each configuration data byte in the **.rpd** file has been organized to ensure the least significant bit (LSB) of each data byte is sent out first to the field-programmable gate array (FPGA) devices during programming. In order to meet this requirement, the LSB of data bytes in the **.rpd** file has to be written first to the EPCS as shown in Figure 7. This format is different from the operation codes and address bytes format.

For operation code and address bytes, SRunner needs to send the most significant bit (MSB) first. Refer to Figure 7.

Figure 7. Timing Diagram During Programming EPCS by Using RPD File as Input



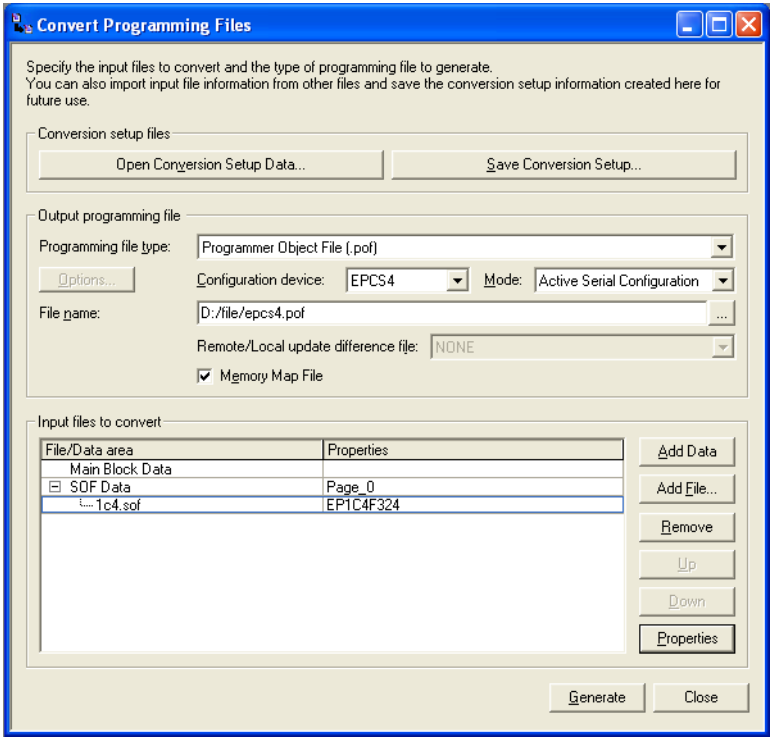
Raw Programming Data (RPD) File

The SRunner requires a **.rpd** file to be used as the input programming file. You need to generate a Programmer Object File (.**pof**), then convert the **.pof** file into a **.rpd** file.

Follow these steps to generate a **.pof**:

1. On the File menu, click **Convert Programming Files**.
2. In the **Convert Programming Files** dialog box, point to the Programming file type menu and click **Programmer Object file**.
3. In the Mode menu, click **Active Serial Configuration**.
4. In the Configuration Device menu, click on a **serial configuration device**.
5. In the **File Name** text box, enter the file name and location for the **.pof** file .
6. In the **Input files to convert** section, point to **SOF Data** and click **Add File** to add the SRAM Object File (**.sof**).
7. Click **Generate** to generate a **.pof** as shown in Figure 8.

Figure 8. Converting an SOF File to a POF File in Convert Programming Files



After compiling the project and generating the **.pof** file, convert the **.pof** file into a **.rpd** file by using the Convert Programming Files dialog box in the File menu. This is shown in Figure 9.

Follow these steps to generate an **.rpd** file from a **.pof** file:

1. On the File menu, click **Convert Programming Files**.
2. In the **Convert Programming Files** window, point to the Programming file type menu and click **Raw Programming Data File.**
3. In the **File Name** text box, enter the file name and location for the **.rpd** file.
4. In the **Input Files to Convert** section, point to **POF** Data and click **Add File** to add the **.pof**.

5. Click **Generate** to generate an **.rpd** file as shown in Figure 9.

Figure 9. Converting a POF File to an RPD File

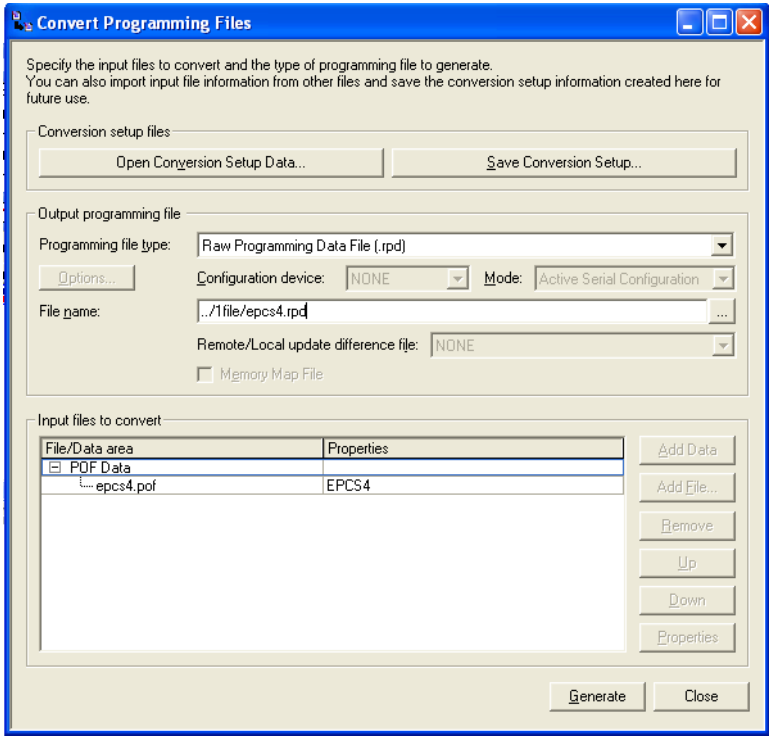


Table 4 shows the file size for the generated **.rpd** file.

Table 4. RPD File Size for Serial Configuration Devices

Device	RPD File Size (bits)
EPCS1	1,048,576
EPCS4	4,194,304
EPCS16	16,777,216
EPCS64	67,108,864
EPCS128	134,217,728

Remove the unused part (blank configuration data) of the **.rpd** file by deleting part of the 0xFF, but leaving at least 36 bytes of 0xFF at the end of the **.rpd** file because they are part of the required configuration data. This helps to reduce the required programmming time. Use a hex editor tool to do the **.rpd** file editing to prevent file corruption.

Pin Assignment

Because the writing and reading of the data to and from the I/O ports on other platforms maps to the parallel port architecture, this document describes the pin assignments of the active serial configuration signals to the parallel port.

These pin assignments reduce the required source code modifications. Figure 5 shows the assignment of the active serial configuration signals to the parallel port.

The parallel port has a total of 12 digital outputs and five digital inputs accessed via three consecutive 8-bit ports as shown in Table 5.

Table 5. Assignment of the Active Serial Configuration

Port	Description	Pins
0	Data Port (labeled with LPT_DATA in source code)	8 output pins
1	Status Port (labeled with LPT_STATUS in source code)	5 input pins (bit 7 inverted)
2	Control Port (labeled with LPT_CONTROL in source code)	4 output pins (bit 0, 1, 3 inverted)

Table 6 shows the pin assignments of the active serial configuration signals to the parallel port.

Table 6. Pin Assignments of Active Serial Configuration Signals to the Parallel Port

Bit	7	6	5	4	3	2	1	0
Port 0	-	ASDI	-	-	nCE	nCS	nCONFIG	DLCK
Port 1	CONF_DONE	-	-	DATAOUT	-	-	-	-
Port 2	-	-	-	-	-	-	-	-

Table 7 describes the function of the active serial configuration pins when you use the SRunner software driver.

Table 7. Pin Description

Pin Name	Description
DCLK	SRunner generate a clock signal to EPCS via the DCLK pin.
ASDI	Data input signal for transferring data serially into the EPCS.
DATAOUT	Data output signal for transferring data serially out of the EPCS to the parallel port during read/programming operation. During read/programming operations, the EPCS is enabled by pulling nCS low.
nCS	The active low chip select input signal toggles at the beginning and end of a valid instruction. When this signal is high, the EPCS is deselected and the DATA pin is tri-stated. When this signal is low, it enables the EPCS and puts the device in an active mode. After power up, the EPCS requires a falling edge on the nCS signal before beginning any operation.
nCE	Active-low chip enable. When nCE is low, the FPGA is enabled. SRunner uses this pin to disable the FPGA from accessing the EPCS during programming.
nCONFIG	Configuration control input. Pulling this pin low during user-mode causes the FPGA to lose its configuration data, enter a reset state, and tri-state all I/O pins. Returning this pin to logic high initiates a reconfiguration. SRunner will hold the FPGA in reset mode by pulling this pin low during programming.
CONF_DONE	This is a dedicated configuration status pin, this pin goes high after the configuration mode is completed.

Porting

The software for SRunner is modular and is portable to other platforms and embedded systems. If needed, you can port the `as.c` file into the new platform and replace the current interfacing module with a platform-specific interfacing module. Different platforms have different ways of interfacing with I/Os and handling data in the memory.

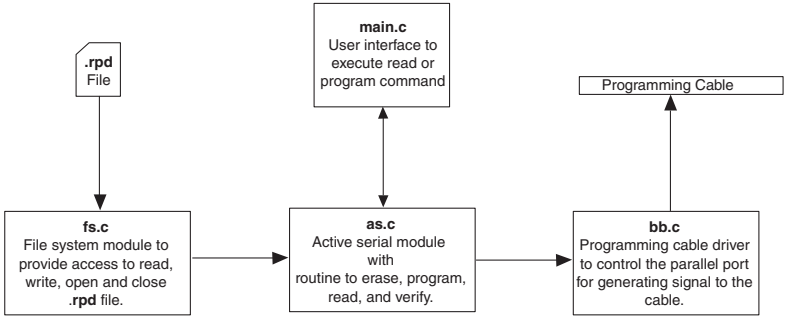
For example, if you need to port source code into an embedded system, some changes are required for the platform-dependent module as shown in Table 8.

Table 8. Porting Change Option

Module	Changes	Platform Dependent
`main.c`	Change to interface with other systems to execute the necessary instruction/operation.	Yes
`bb.c`	Change to interface to either programming cable or directly to the EPCS Pin assignment (reassign pins to other I/O pins).	Yes
`fs.c`	Change to access to the programming file (that is stored in other memory).	Yes
`as.c`	Some minor changes if required.	No

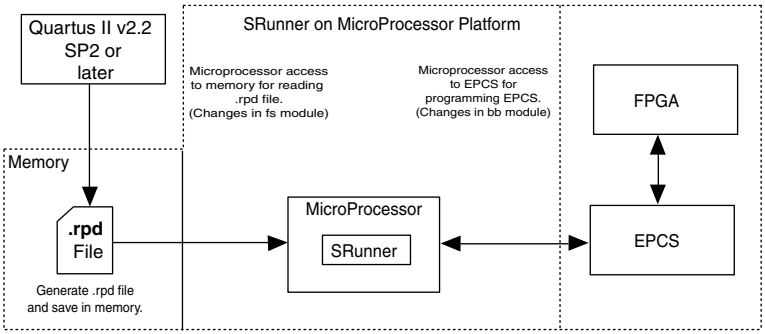
Figure 10 shows the changes required when you port the SRunner into other platforms.

Figure 10. Changes Required for Porting SRunner into Other Platforms



You can also port the SRunner into an embedded system as shown in Figure 11.

Figure 11. Reference for Porting SRunner into an Embedded System



Conclusion

The SRunner is a standalone software driver developed for EPCS programming and can be customized to fit in other systems and embedded systems. The customizable driver allows fast and easy EPCS programming using other platform-like microprocessor and tester programs.

Referenced Documents

- *Serial Configuration Devices (EPCS1, EPCS4, EPCS16, EPCS64 and EPCS128) Data Sheet*.

Document Revision History

Table 9 shows the revision history for this chapter.

Table 9. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
June 2008 v 1.1	• Added new bullet 'Verify EPCS data' in the "Features" section. • Updated Table 1. • Updated Figures 2, Figure 3, Figure 4 and Figure 6. • Added note to Figure 6. • Added another row for EPC128 in Table 4 .	—
October 2006 v 1.0	• Initial Release.	—

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
literature@altera.com