



XAPP058 (v3.1) June 25, 2004

Xilinx In-System Programming Using an Embedded Microcontroller

Summary

The Xilinx high-performance CPLD, FPGA, and configuration PROM families provide in-system programmability, reliable pin locking, and JTAG boundary-scan test capability. This powerful combination of features allows designers to make significant changes and yet keep the original device pinouts, thus, eliminating the need to re-tool PC boards. By using an embedded controller to program these CPLDs and FPGAs from an on-board RAM or EPROM, designers can easily upgrade, modify, and test designs, even in the field.

Xilinx Families

XC9500, XC9500XL, XC9500XV, XC4000, XC18V00, CoolRunner™, Spartan™, Virtex™

Introduction

The Xilinx CPLD and FPGA families combine superior performance with an advanced architecture to create new design opportunities that were previously impossible. The combination of in-system programmability, reliable pin locking, and JTAG test capability gives the following important benefits:

- Reduces device handling costs and time to market
- Saves the expense of laying out new PC boards
- Allows remote maintenance, modification, and testing
- Increases the life span and functionality of products
- Enables unique, customer-specific features

The ISP controller shown in **Figure 1** can help designers achieve these unprecedented benefits by providing a simple means for automatically programming Xilinx CPLDs and FPGAs from design information stored in EPROM. This design is easily modified for remote downloading applications and the included C-code can be compiled for any microcontroller.

To create device programming files, Xilinx provides the JTAG Programmer™ software that automatically reads standard JEDEC/BIT/MCS/EXO device programming files and converts them to SVF format, which contains both data and programming instructions for the CPLDs, FPGAs, and configuration PROMs; it reads JEDEC files for CPLDs, BIT files for FPGAs, and MCS/EXO files for configuration PROMs. These files are then converted to a compact binary format (XSVF) and can be stored in the on-board EPROM. The 8051 microcontroller interprets the XSVF information and generates the programming instructions, data, and control signals for the Xilinx devices.

By using a simple IEEE 1149.1 (JTAG) interface, Xilinx devices are easily programmed and tested without using expensive hardware. Multiple devices can be daisy-chained, permitting a single 4-wire Test Access Port (TAP) to control any number of Xilinx devices or other JTAG-compatible devices.

The files and utilities associated with this application note are available in a package for downloading from ftp://ftp.xilinx.com/pub/swhelp/cpld/eisp_pc.zip

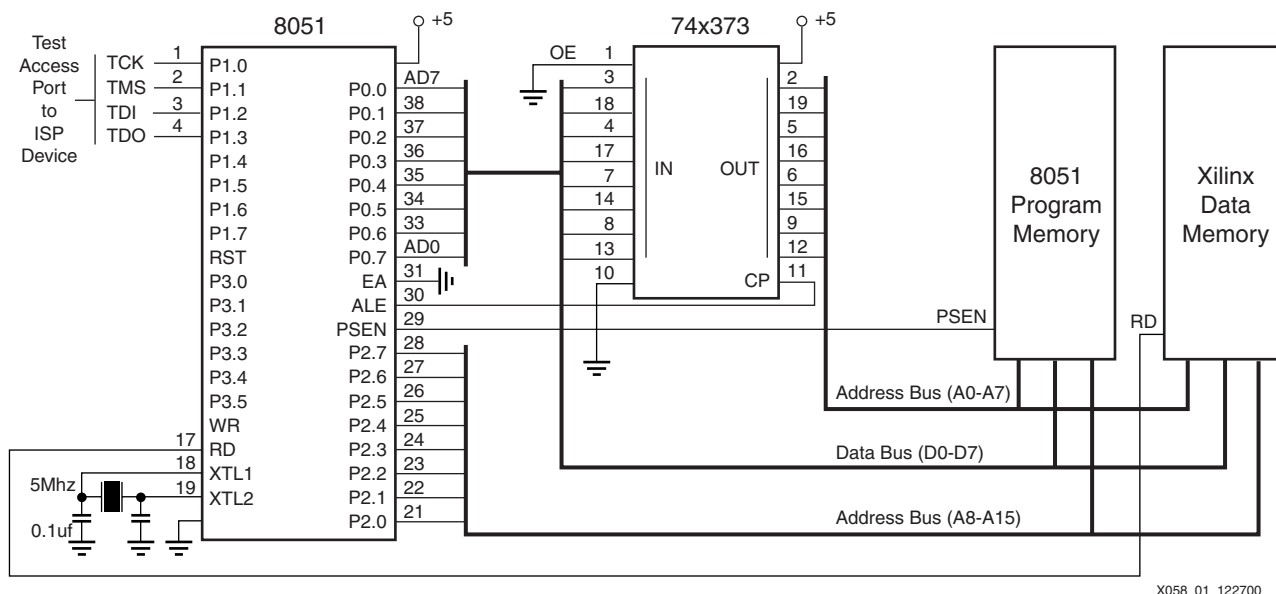


Figure 1: ISP Controller Schematic

Programming Xilinx CPLDs, FPGAs, and Configuration PROMs

Serial Vector Format (SVF) is a syntax specification for describing high level IEEE 1149.1 (JTAG) bus operations. SVF was developed by Texas Instruments and has been adopted as a standard for data interchange by JTAG test equipment and software manufacturers such as Teradyne, Tektronix, and others. Xilinx CPLDs, FPGAs, and configuration PROMs accept programming and JTAG boundary-scan test instructions in SVF format, via the TAP. The timing for these TAP signals is shown in [Figure 17, page 19](#).

The JTAG Programmer software automatically converts standard JEDEC/BIT/MCS/EXO programming files into SVF format. However, the SVF format is ASCII which is inefficient for embedded applications due to its memory requirements. Therefore, to minimize the memory requirements, SVF is converted into a more compact (binary) format called XSVF. In this design, an 8051 C-code algorithm interprets the XSVF file and provides the required JTAG TAP stimulus to the CPLD, performing the programming and (optional) test operations which were originally specified in the SVF file.

Notes:

1. For a description of the SVF and XSVF commands and file formats, see [Appendix A: SVF File Format for Xilinx Devices, page 24](#) and [Appendix B: XSVF File Format and Conversion Utilities, page 26](#).

The flow for creating the programming files that are used with this design, is shown in [Figure 2](#).

JTAG Instruction Summary

Xilinx devices accept both programming and test instructions using the JTAG TAP. The JTAG commands and descriptions used for programming and functional testing are as follows:

Instructions Supported by All Devices

- **EXTEST** - Isolates the device I/O pins from the internal device circuitry to enable connectivity tests between devices. It uses the device pins to apply test values and to capture the results.
- **INTEST** - Isolates the device from the system, applies test vectors to the device input pins, and captures the results from the device output pins.
- **SAMPLE/PRELOAD** - Allows values to be loaded into the boundary scan register to drive the device output pins. Also captures the values on the input pins.
- **BYPASS** - Bypasses a device in a boundary scan chain by functionally connecting TDI to TDO.

Instructions Common to CPLD, FPGAs, and Configuration Proms

- **EXTEST** - Isolates the device I/O pins from the internal device circuitry to enable connectivity tests between devices. It uses the device pins to apply test values and to capture the results.
- **IDCODE** - Returns a 32-bit hardwired identification code that defines the part type, manufacturer, and version number.
- **HIGHZ** - Causes all device pins to float to a high impedance state.

Instructions Supported by XC4000/Spartan Only

- **CONFIGURE** - Allows access to the configuration bus for configuration.
- **READBACK** - Allows access to the configuration bus for readback.

Instructions Supported by Virtex Only

- **CFG_IN/CFG_OUT** - Allows access to the configuration bus for configuration and readback.
- **JSTART** - Clock the startup sequence when startup clock = JTAGCLK.

Commands Supported by CPLDs and Configuration PROMs

- **ISPEN** - Enables the ISP function in the XC9500/XL/XV device, floats all device function pins, and initializes the programming logic.
- **FERASE** - Erases a specified program memory block.
- **FPGM** - Programs specific bit values at specified addresses. An FPGMI instruction is used for the XC95216 and larger devices which have automatic address generation capabilities.
- **FVIFY** - Reads the fuse values at specified addresses. An FVFI instruction is used for the XC95216 and larger devices which have automatic address generation capabilities.
- **ISPEX** - Exits ISP Mode. The device is then initialized to its programmed function with all pins operable.

The following instructions are also available but are not used for programming or functional testing:

Instructions Specific to CPLDs and Configuration PROMs

- **USERCODE** - Returns a 32-bit user-programmable code that can be used to store version control information or other user-defined variables.

Instructions Specific to XC4000/Spartan

- **USER1/USER2** - These instructions allow capture, shift and update of user-defined registers.

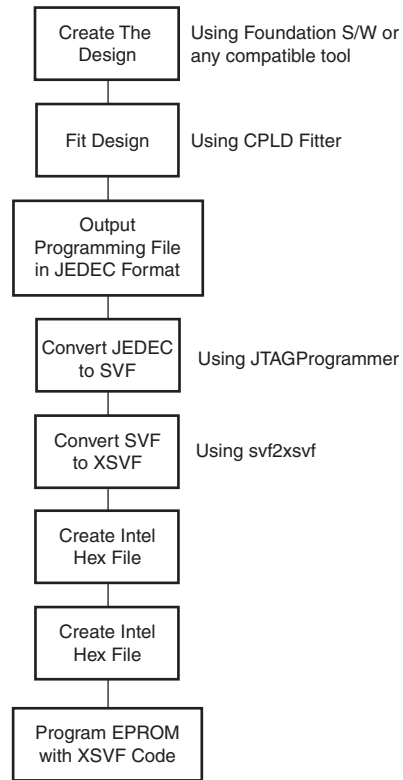
Instructions Specific to Virtex

- **USR1/USR2** - These instructions allow capture, shift and update of user-defined registers.

Instructions Specific to Configuration PROMs

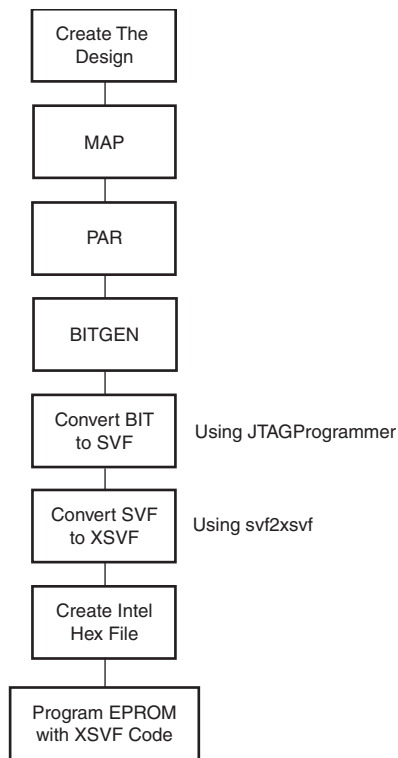
- **FADDR** - Sets the PROM array address register.
- **DATA0** - Accesses the array word-line register.
- **PROGRAM** - Programs the word-line into the array.
- **SERASE** - Globally refines the programmed values in the array.

The programming flow charts for CPLDs, FPGAs, and Configuration PROMs are shown in [Figure 2](#), [Figure 3](#), and [Figure 4](#), respectively.



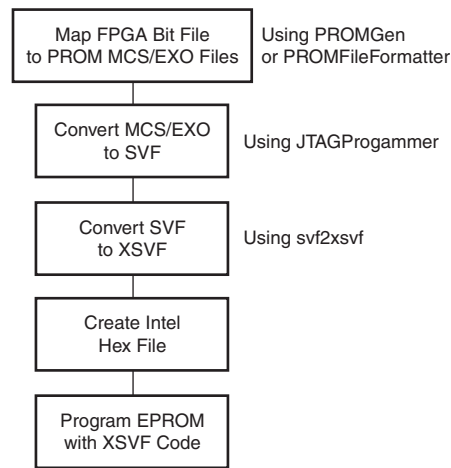
X058_02_071101

Figure 2: CPLD Program Flow



X058_03_071101

Figure 3: FPGA Program Flow



X058_04_122700

Figure 4: Configuration PROM Flow

Creating an SVF File Using JTAG Programmer

This procedure describes how to create an SVF file; it assumes that Xilinx Foundation or Alliance WebPACK series software version 3.1 or newer is being used. These software packages include the Xilinx CPLD fitter, FPGA mapping tool, and JTAG Programmer software. JTAG Programmer is supplied with both a graphical and batch user interface. The batch user interface executable is typically named “jtagprog” and the graphical user interface is named “jtagpgmr.” The graphical tool is always launched from the Design Manager or Project Manager. The batch tool is available by opening a shell and invoking “jtagprog” on the command line.

Using the batch download tool to generate SVF files.

1. Fit the design and create a JEDEC/BIT programming file.
2. Invoke the batch JTAG Programmer tool from the command line in a new shell:

```
jtagprog -svf
```

The following messages appear:

```
JTAGProgrammer: version <Version Number>
Copyright: 1991-1998
```

```
Sizing system available memory...done.
```

```
*** SVF GENERATION MODE ***
```

```
[JTAGProgrammer: (1)] >
```

3. Set up the device types and assign design names. To do this type following command at the JTAG Programmer prompt:

```
part deviceType1:designName1 deviceType2:designName2 ...
deviceTypeN:designNameN <CR>
```

where deviceType is the name of the BSDL file without the **.bsd** extension for that device and designName is the name of the design to translate into SVF. Multiple deviceType:designName pairs are separated by spaces. For example:

```
part xc95108:abc12 xc18V04:ww133 xcv50.pg240_efg
```

The “part” command defines the composition and ordering of the boundary-scan chain. The devices are arranged with the first device specified being the first to receive TDI information and the last device being that which provides the final TDO data.

Notes:

1. For any non-Xilinx devices in the boundary-scan chain, make certain that the BSDL file is available either in the XILINX variable data directory or by specifying complete path information in the deviceType. The designName in this case can be any arbitrary name. Alternatively, non-Xilinx devices in the boundary-scan chain can be specified using the following syntax for the device type, where # is the length of the device's instruction register:

```
register-s#
```

4. Execute the required boundary-scan or ISP operation in JTAG Programmer:
 - a. **erase [-fh] designName** – generates an SVF file to describe the boundary-scan sequence to erase the specified part. The `-f` flag is used to generate an erase sequence that overrides write protection on devices. The `-h` flag is used to specify that all other parts (i.e., not designName) in the boundary-scan chain should be held in the HIGHZ state during the erase operation.
 - b. **verify [-h] designName [-j jedecFileName]** – generates an SVF file to describe the boundary-scan sequence to read back the device contents and compare it against the contents of the specified JEDEC file. The JEDEC file defaults to be designName.jed in the current directory or can be alternatively specified using the `-j` flag. The `-h` flag is used to specify that all other parts (i.e., not designName) in the boundary-scan chain should be held in the HIGHZ state during the verify operation.
 - c. **program [-bhv] designName -j [jedecFileName/mcsFileName/exo/FileName/bitFileName]** - generates an SVF file to describe the boundary-scan sequence to program the device using that programming data specified JEDEC/BIT/MCS/EXO file. The JEDEC/BIT/MCS/EXO file defaults to be designName.jed or designName.bit in the current directory or can be alternatively specified using the `-j` flag. The `-h` flag is used to specify that all other parts (i.e., not designName) in the boundary-scan chain should be held in the HIGHZ state during the programming operation. The `-b` flag instructs the programming operations to skim the erase operation for the device. This is useful when programming devices shipped from the factory which are always delivered blank. The `-v` flag instructs the programmer to include the verify operation after programming.
 - d. **partinfo [-h] designName -id** - generates an SVF file to describe the boundary-scan sequence to read back the 32 bit hard-coded device IDCODE. The `-h` flag is used to specify that all other parts (i.e., not designName) in the boundary-scan chain should be held in the HIGHZ state during the IDCODE operation.
 - e. **partinfo [-h] designName -signature** - generates an SVF file to describe the boundary-scan sequence to read back the 32-bit user-programmed device USERCODE. The `-h` flag is used to specify that all other parts (i.e., not designName) in the boundary-scan chain should be held in the HIGHZ state during the USERCODE operation.

Notes:

1. The recommended command for programming a CPLD or configuration PROM is:

```
program -h -v designName -j [jedecFileName/mcsFileName/exoFileName]
```
2. The recommended command for programming an FPGA is:

```
program -h designName -j bitFileName
```

5. Exit JTAG Programmer by entering the following command:

```
Quit
```

Notes:

1. The SVF file is named designName.svf and is created in the current working directory. Consecutive operations on the same designName appends to the SVF file. To create SVF files with separate operations in each, rename the SVF file after each operation by exiting to the system shell.

Using the graphical user interface to generate SVF files:

1. Fit the design and create a JEDEC programming file.
2. Double-click on the JTAG Programmer icon, or open a system shell and type “jtagpgmr.” The JTAG Programmer appear as shown in [Figure 5](#).

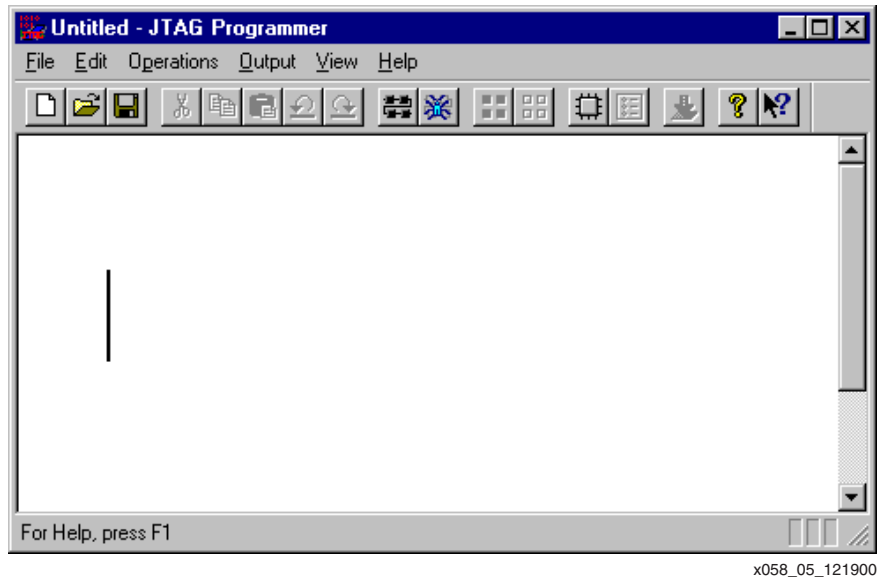


Figure 5: JTAG Programmer

3. Instantiate the boundary-scan chain in either of two ways. The first method is as follows:
 - a. Manually add each device in the correct boundary-scan order from system TDI to system TDO.
 - b. Select Edit->Add device for each device as it exists in the boundary-scan chain in [Figure 6](#).

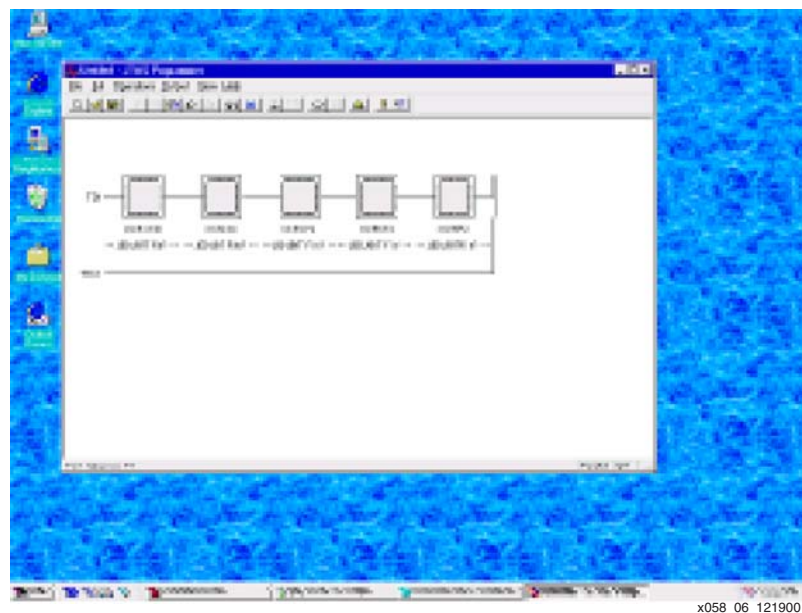
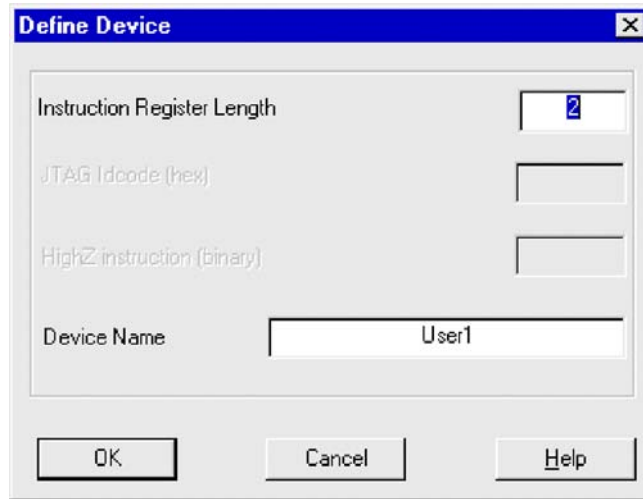


Figure 6: Add Device

- c. Fill in the device properties dialog to identify the JEDEC/BIT/MCS/EXO file (if it is a Xilinx device) or BSDL file (if it is not an Xilinx device) associated with the device being

added. Non-Xilinx devices can be specified using the Edit->Define Device menu item. (See bitmap of Edit ->Define Device dialog.) Enter the device's instruction register length into the Define Device dialog box. See [Figure 7](#).



x058_07_121900

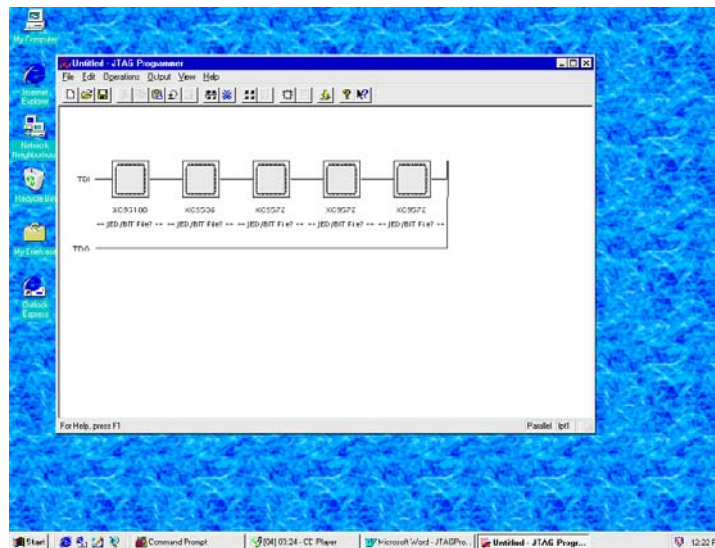
Figure 7: Define Device Dialog

Notes:

1. The device type and JEDEC, BIT, MCS, and EXO file name appears below the added device. The second method is to allow JTAG Programmer to query the boundary-scan chain for devices and their ordering and then fill in the JEDEC and BSDL file information. This method works only when the target system is connected to your computer and powered up. The steps are as follows:

- a. Select File->Initialize chain.
- b. Perform no operations on the devices other than those specified.

JTAG Programmer displays the boundary-scan chain configuration as shown in [Figure 8](#).



x058_08_121900

Figure 8: Boundary-Scan Chain

- c. For each device in the resulting chain, double click on the chip icon to bring up the device properties dialog and select the JEDEC, BIT, MCS, EXO, or BSDL file associated with that device.
- d. Put the JTAG Programmer into SVF mode by selecting Output->Create SVF file... to create a new SVF file or Output->Append to SVF file... to append to an existing SVF file. Fill in the SVF file dialog with the desired name of the target SVF file to be created.

Notes:

1. Once the SVF mode is entered, the composition of the boundary-scan chain cannot be edited in order to ensure consistency of the boundary-scan data in the SVF file.
- e. Select at least one device on which to operate by clicking on the chip icon. the icon should become highlighted when it is selected. To select more than one device, hold down the shift key while clicking on additional devices.
- f. Invoke the items in the Operations menu to generate corresponding command in the SVF file for the selected devices.

Notes:

1. The recommended operation for programming selected devices is Operations ->Program with the "Erase before Programming" and "Verify" check boxes selected. The "Verify" check box is not recommended for FPGA devices.
- g. When the required operations are complete, exit the JTAG Programmer by selecting File->Exit.

Notes:

1. The "Use HIGHZ instead of BYPASS" option from the File->Preferences... dialog can be selected to specify that all other parts (i.e., not the device selected) in the boundary-scan chain should be held in the HIGHZ state during the requested operation.

To generate separate SVF files for each operation, perform the following steps between operations:

1. Select Output->Use Cable
2. On the Cable Communications Dialog select Cancel
3. Select Output->Create SVF File
4. Choose a new SVF file and proceed normally.

EPROM Programming

To program an EPROM, the binary XSVF file must be converted to an Intel Hex or similar PROM format file. Most embedded processor development system software automatically converts included binary files to the appropriate format.

Software Limitations

JTAG Programmer can generate SVF files only for devices for which JEDEC/BIT/MCS/EXO files can be created. Designers should verify that the development software they are using can create JEDEC/BIT/MCS/EXO files for the specific devices they intend to use.

For instructions on generating SVF for CoolRunner CPLDs, go to the Xilinx Support website at <http://support.xilinx.com> and search for answer 7565.

Hardware Design

As shown in [Figure 1, page 2](#), this design requires only an 8051 microcontroller, an address latch, and enough EPROM or RAM to contain both the 8051 code and the CPLD/FPGA/PROM programming data.

Hardware Design Description

The example 8051 allows 64K of program and 64K of data space; however, some devices require more data space.

The 8051 multiplexes port 0 for both data and addresses. The ALE signal causes the 74x373 to latch the low order address, and the high order address is output on port 2. Port 0 then floats, allowing the selected EPROM to drive the data inputs. Then the PSEN signal goes low to activate an 8051 program read operation, or the \overline{RD} signal goes low to activate a CPLD programming data read operation.

Estimated EPROM Memory Requirements

[Table 1](#) shows the estimated EPROM capacity needed to contain the programming data.

Table 1: XSVF File Sizes

Device Type	File Size (bytes)
XC9536	45572
XC9572	103928
XC95108	175250
XC95144	144222
XC95216	259620
XC95288	403698
XC9536XL	38186
XC9572XL	51590
XC95144XL	78398
XC95288XL	132014
XCR3064XL	21149
XCR3128XL	40067
XCR3256XL	90042
XC18V512	338119
XC18V01	675399
XC18V02	1341767
XC18V04	2682183
XCS20XL	24010
XCS40XL	44186
XC2S100	103969
XC2S150	138352
XCV300	232876
XCV1000	814055
XCV100E	114943
XCV300E	249318
XCV600E	526368
XCV1000E	875119
XCV2000E	1349542

The XSVF file sizes are dependent only on the device type, not on the design implementation. If further compression of the XSVF file is needed, a standard compression technique, such as Lempel-Ziv can be used.

Modifications for Other Applications

The design presented in this application note is for a stand-alone ISP controller. However, it is also possible to apply these techniques to microcontrollers that might already exist within a design. To implement this design in an already existing microcontroller, all that is needed is four I/O pins to drive the TAP, and enough storage space to contain both the controller program and the CPLD/FPGA/PROM download data. In addition, care must be taken to preserve the JTAG port timing.

The TAP timing in this design is dependent on the 8051 clock. For other 8051 clock frequencies or for different microcontrollers, the timing must be calculated accordingly, in order to implement the timing specified in [Exception Handling, page 21](#).

The speed at which the TAP ports can be toggled affects the overall programming time for FPGAs and PROMs that require millions of TCK cycles to shift just the data. For CPLDs, the cumulative program pulse time has a greater affect on programming time than the data shift time.

Using a different microcontroller would require changing the I/O subroutine calls while preserving the correct TAP timing relationships. These subroutine calls are located in the ports.c file. All other C-code is independent of the microcontroller and does not need to be modified.

RAM can be used instead of the EPROM in this design. This would allow the CPLD/FPGA/PROM devices to be programmed and tested remotely via modem, using remote control software written by the user.

Debugging Suggestions

The following suggestions can be helpful in testing this design:

- View the contents of the XSVF file using the -a option for the svf2xsvf converter. This option generates a text file version of the xsvf.
- Decrease the TCK frequency to test that the wait times for program and erase are sufficiently long.
- Make certain that the function pins go into a 3-state condition in ISP mode.
- Test that the function pins initialize when ISP mode is terminated with the ISPEX command.
- Verify that the devices which are not being programmed are in bypass mode. Bypass mode causes TDO to be the same as TDI, delayed by one TCK clock pulse.
- Use the precompiled playxsvf.exe from the download package to execute the XSVF on a PC through the Parallel Cable.
- Generate a simple XSVF that only checks the IDCODE of the target device to test basic functionality of the hardware and software.
- Generate and execute separate XSVF files for the erase, blank check, program, and verify operations to narrow the problem area.
- Program the device from JTAG Programmer and a download cable to verify basic hardware functionality.

Firmware Design

The flow chart for the C-Code is shown in [Figure 9](#) through [Figure 16](#). This code continuously reads the instructions and arguments from the XSVF file contained in the program data EPROM and branches in one of three ways based on the three possible XSVF instructions (XRUNTEST, XSIR, XSDR) as described in [Appendix B: XSVF File Format and Conversion Utilities, page 26](#).

When the C-Code reads an XRUNTEST instruction, it reads in the next four bytes of data that specify the number of microseconds for which the device stays in the Run-Test/Idle state before the next XSIR or XSDR instruction is executed. The runTestTimes variable is used to store this value.

When the C-Code reads an XSIR instruction, it provides stimulus to the TMS and TCK ports until it arrives in the Shift-IR state. It then reads a byte that specifies the length of the data and the actual data itself, outputting the specified data on the TDI port. Finally, when all the data has been output to the TDI port, the TMS value is changed and successive TCK pulses are output until the Run-Test/Idle state is reached again.

When the C-Code reads an XSDR instruction, it reads the data specifying the values that are output during the Shift-DR state. The code then toggles TMS and TCK appropriately to transition directly to the Shift-DR state. It then holds the TMS value at 0 in order to stay in the Shift-DR state and the data from the XSVF file is output to the TDI port while storing the data received from the TDO port. After all the data has been output to the TDI port, TMS is set to 1 in order to move to the Exit-1-DR state. Then, the TDO input value is compared to the TDO expected value. If the two values fail to match, the exception handling procedure is executed as shown in [Figure 19, page 21](#). If the TDO input values match the expected values, the code returns to the Run-Test/Idle state and waits for the amount of time specified by the runTestTimes variable (which was originally set in the XRUNTEST instruction).

Memory Map

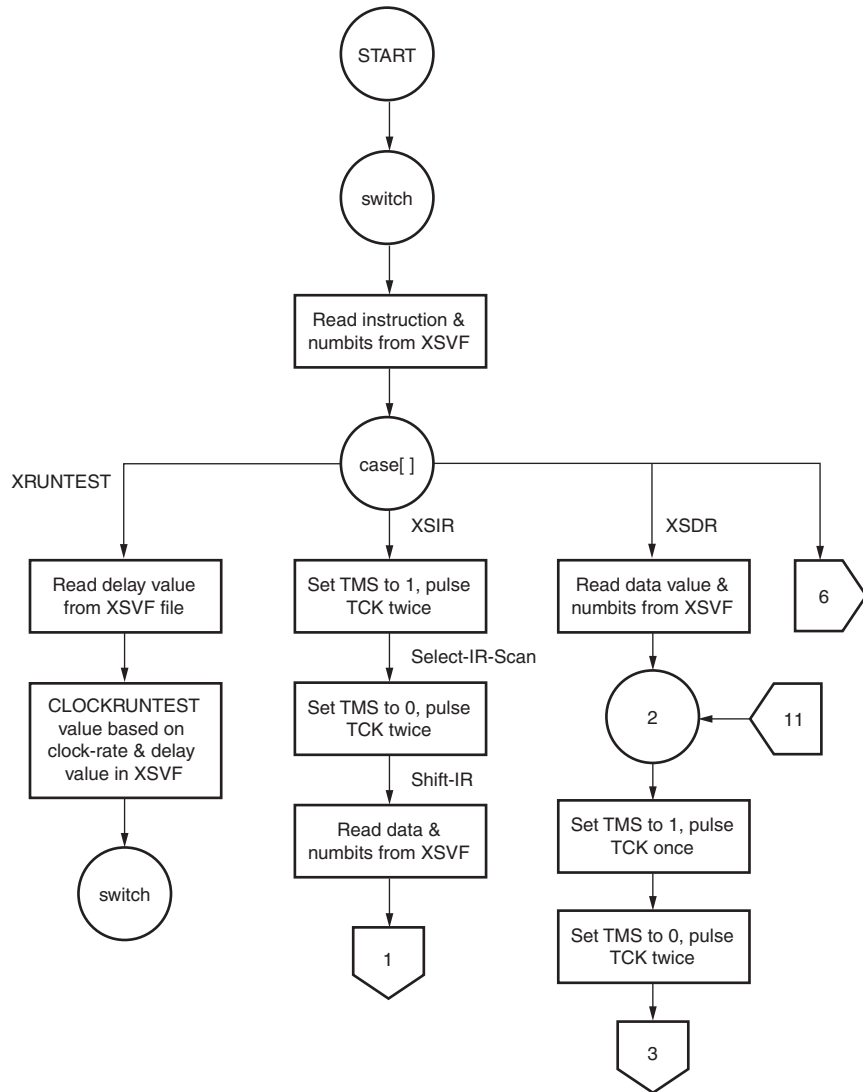
The 8051 memory map is divided into two 64K byte blocks: one for the 8051 program and one for data. The 8051 program memory resides in the 8051 program block and is enabled by the $\overline{\text{PSEN}}$ signal. The Xilinx PLD program memory resides in the 8051 data block and is enabled by the $\overline{\text{RD}}$ signal. When additional data space is required, use one of the methodologies specified in the specific microprocessor's applications note.

Port Map

The 8051 I/O ports are used to generate the memory address and the TAP signals, as shown in [Figure 1, page 2](#). Port 1 of the 8051 is used to control the TAP signals; [Table 2](#) shows the port configuration.

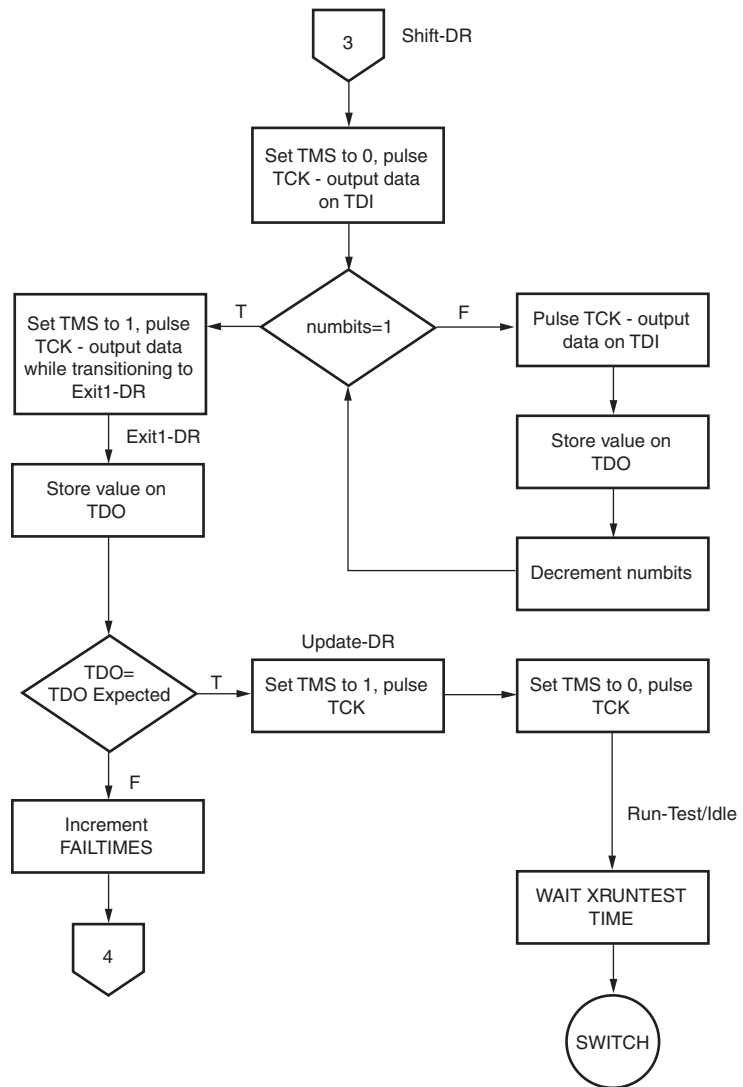
Table 2: 8051 Port 1 Mapping

TAP Pin	Port1 Bit	Configured As
TCK	0	Input
TMS	1	Input
TDI	2	Input
TDO	3	Output



X058_08_122700

Figure 9: Flow Chart for the ISP Controller Code

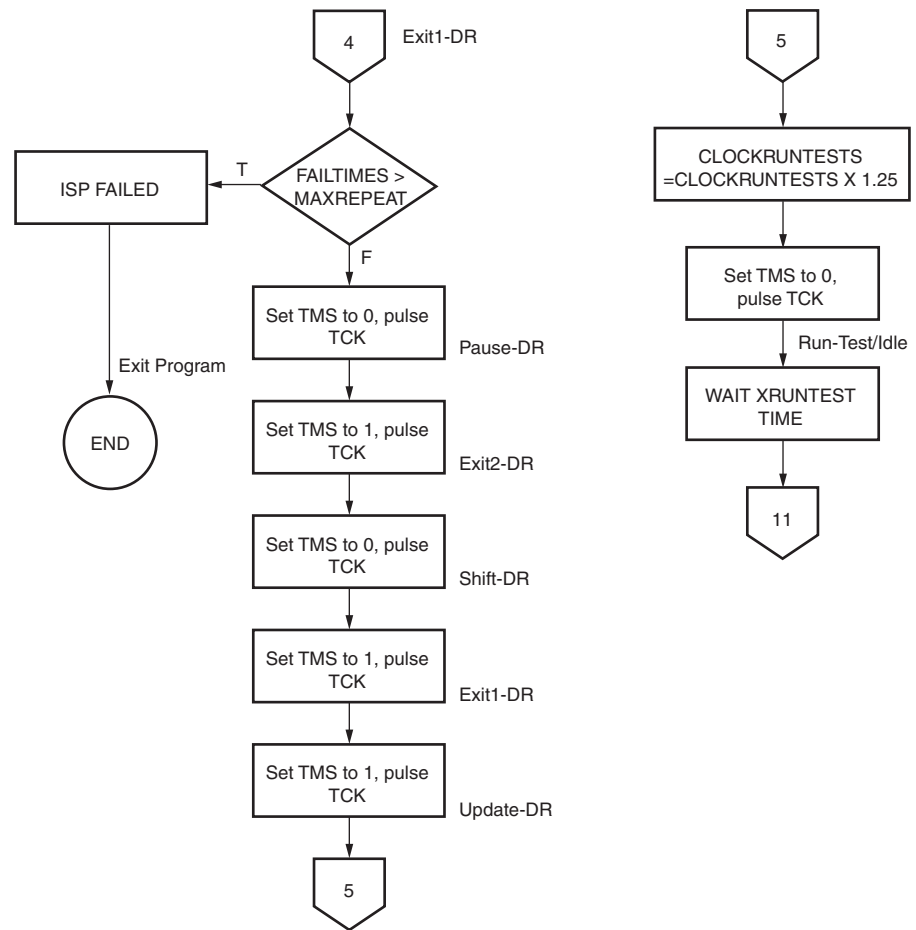


X058_10_010901

Figure 10: Flow Chart for the ISP Controller Code (Continued)

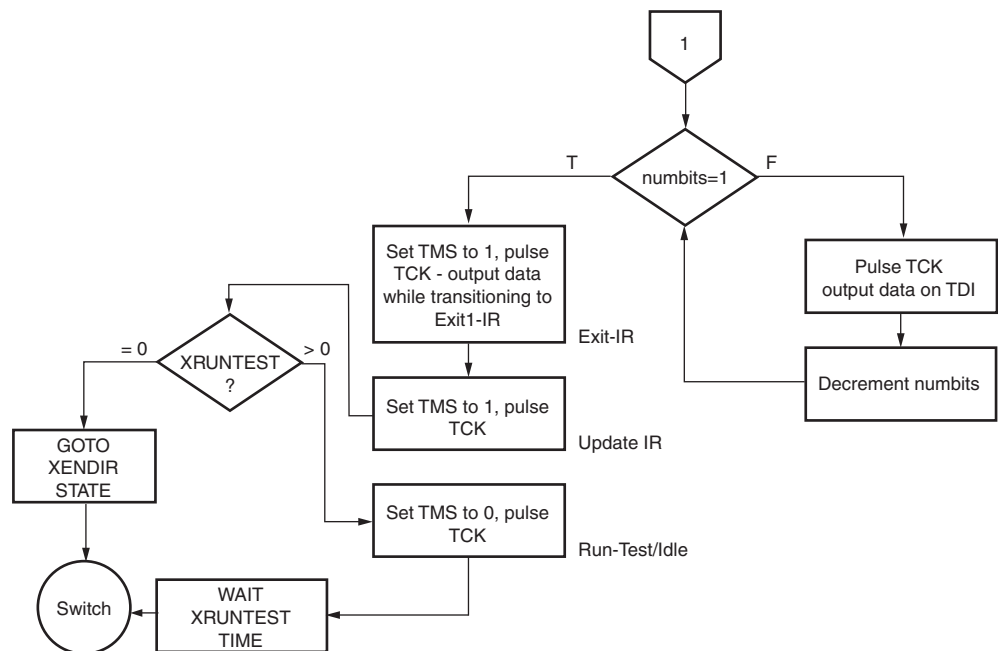
Notes:

1. For FPGAs, step 4 is scrapped completely if the TDO expected does not match the actual TDO; the program quits with an error message.



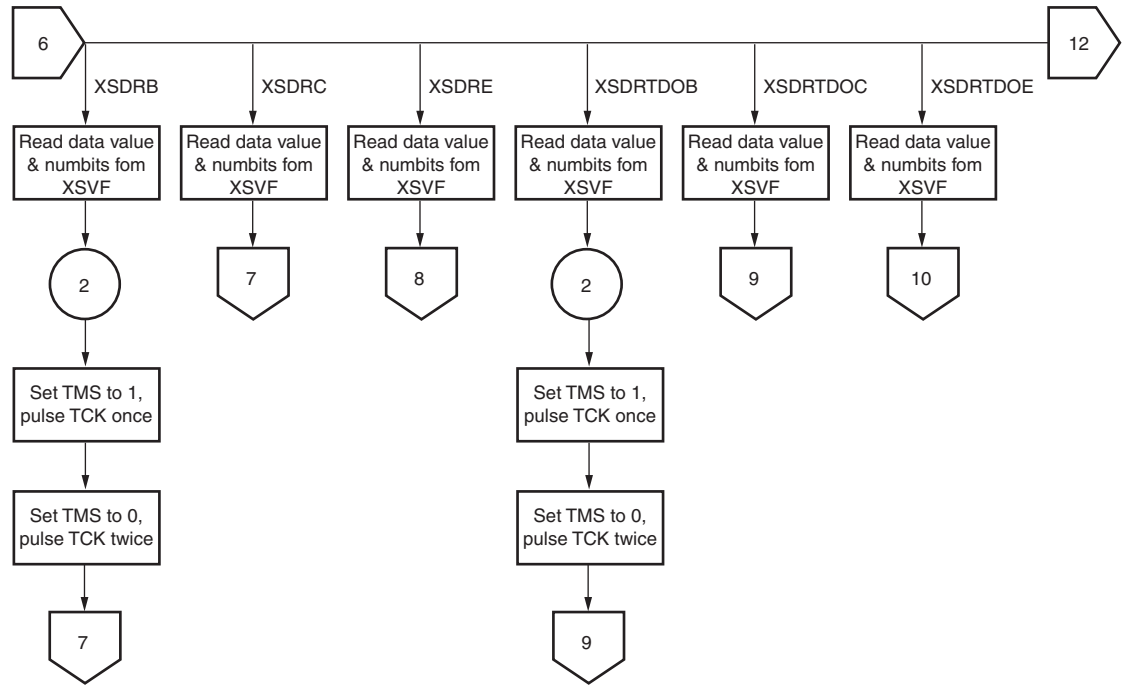
X058_11_010901

Figure 11: Flow Chart for the ISP Controller Code (Continued)



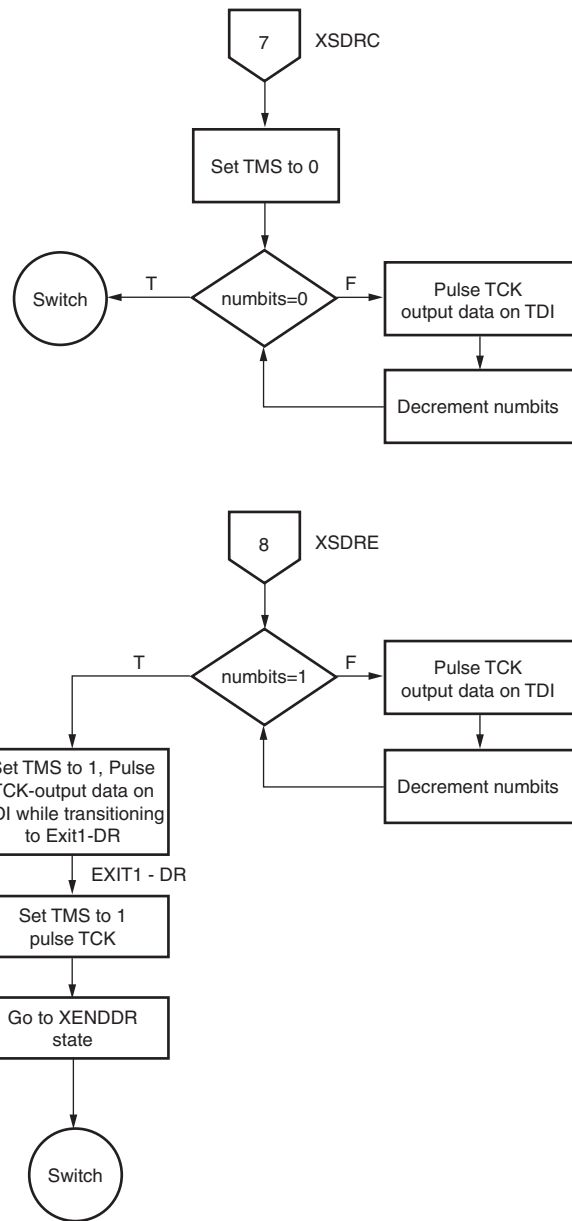
X058_12_010901

Figure 12: Flow Chart for the ISP Controller Code (Continued)



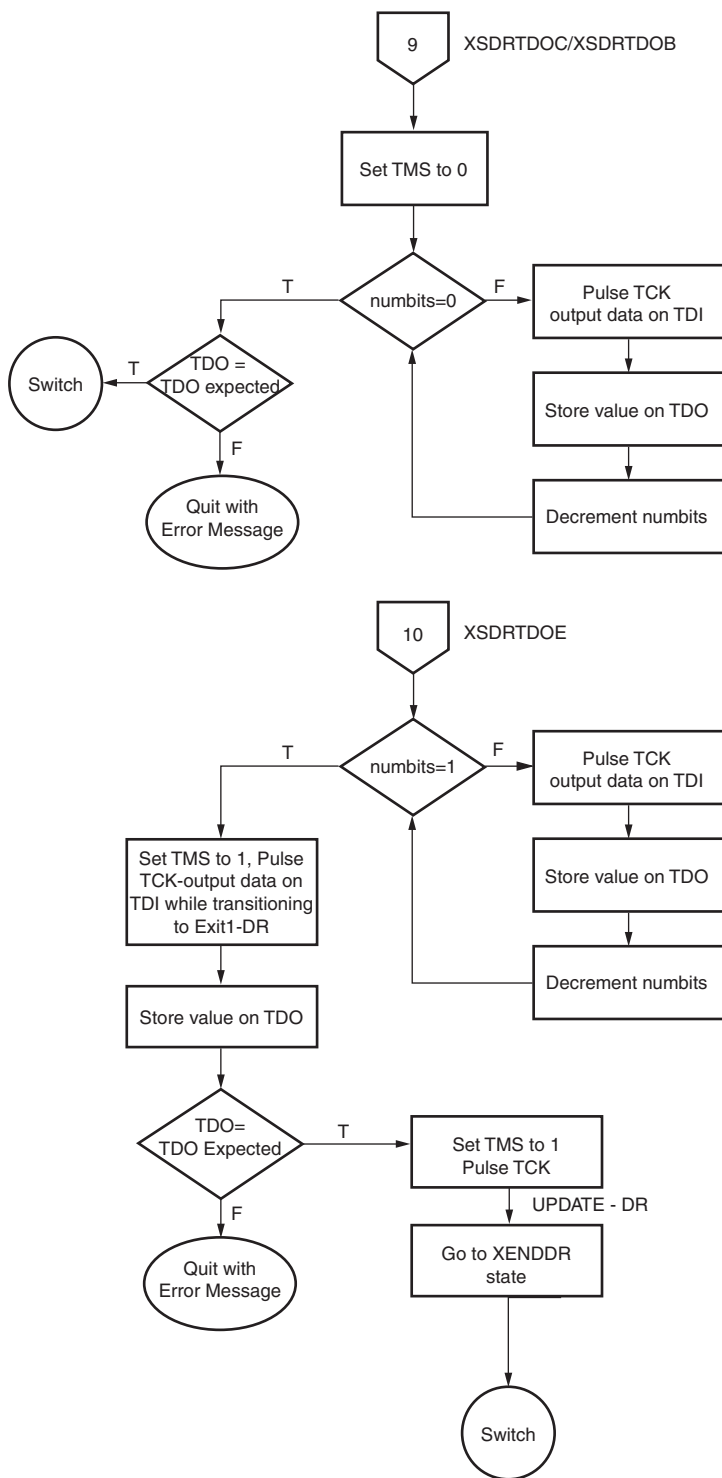
X058_13_122100

Figure 13: Flow Chart for the ISP Controller Code (Continued)



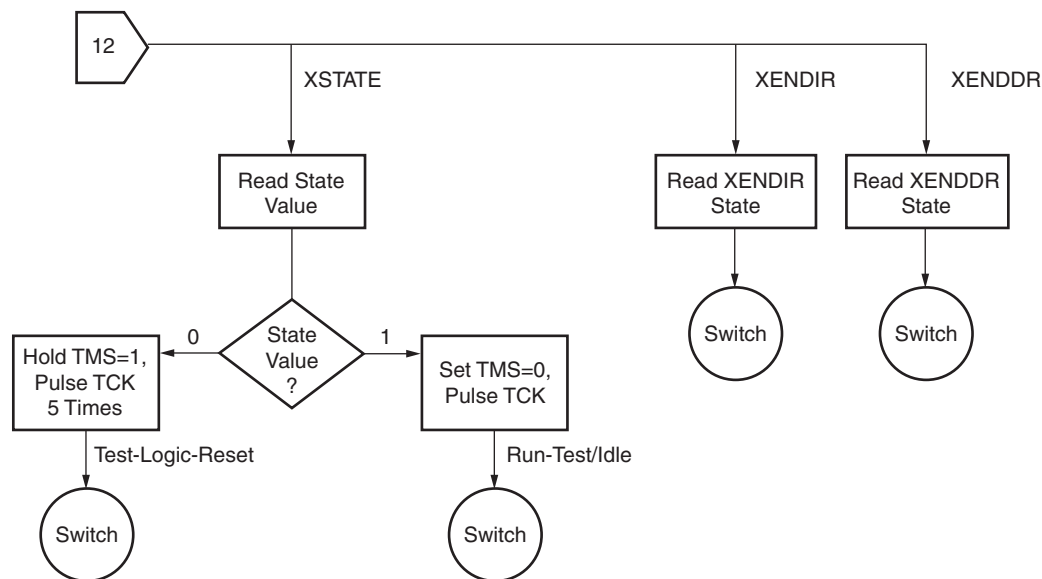
X058_14_010901

Figure 14: Flow Chart for the ISP Controller Code (Continued)



X058_15_011201

Figure 15: Flow Chart for the ISP Controller Code (Continued)

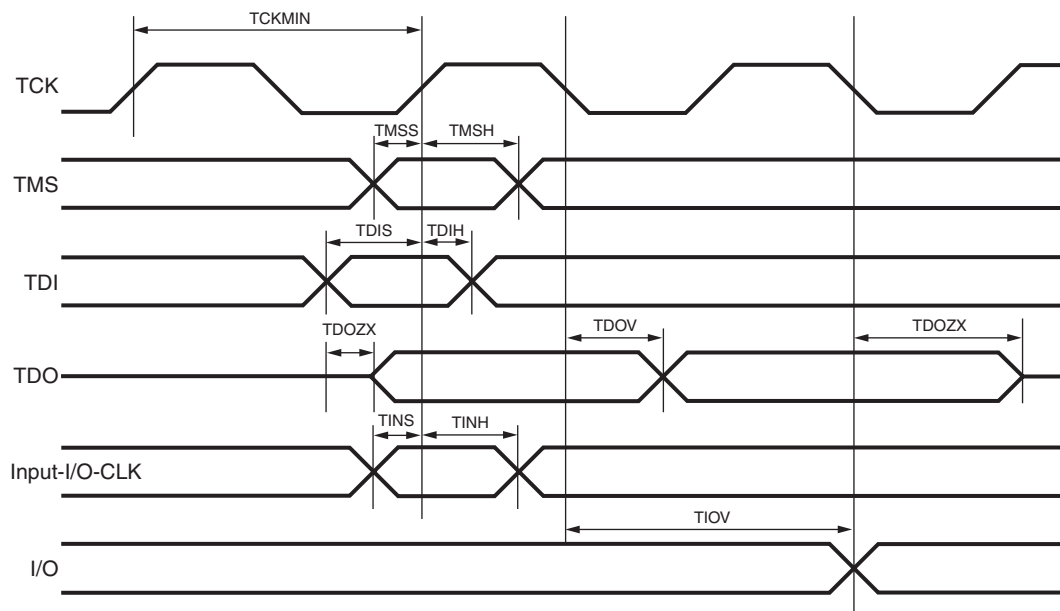


X058_16_010901

Figure 16: Flow Chart for the ISP Controller Code (Concluded)

TAP Timing

Figure 17 shows the timing relationships of the TAP signals. The C-code running on the 8051 insures that the TDI and TMS values are driven at least two instruction cycles before asserting TCK. At that same time, TDO can be strobed.



X058_18_122100

Figure 17: Test Access Port Timing

The key timing relationships include:

- TMS and TDI are sampled on the rising edge of TCK.
- A new TDO value appears after the falling edge of TCK.

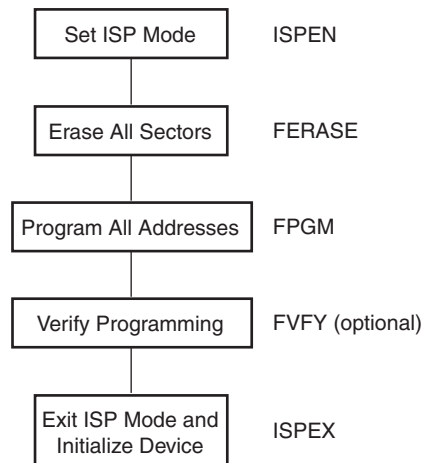
The C-code ensures proper TAP timing by:

- Updating TMS and TDI on the falling edge of TCK
- Sampling TDO after a sufficient delay following the falling edge of TCK.

Parts of the XSVF file specify wait times during which the device programs or erases the specified location or sector. Implementation of the wait timer can be accomplished either by software loops that depend on the processor's cycle time or by using the 8051's built-in timer function. In this design, timing is established through software loops in the ports.c file.

Figure 18 shows the XC9500/XL/XV device programming flow.

Table 3 lists the XC9500 timing parameters for the TAP waveforms shown in Figure 17. For other device families, see the device family data sheet for TAP timing characteristics.



X058_17_122100

Figure 18: XC9500/XL/XV Device Programming Flow

Table 3: XC9500 Test Access Port Timing Parameters (ns)

Symbol	Parameter	Min	Max
TCKMIN	TCK Minimum Clock Period	100	
TMSS	TMS Setup Time	10	
TMSH	TMS Hold Time	10	
TDIS	TDI Setup Time	15	
TDIH	TDI Hold Time	25	
TDOZX	TDO Float to Valid Delay		35
TDOXZ	TDI Valid to Float Delay		35
TDOV	TDO Valid Delay		35
TINS	I/O Setup Time	15	
TINH	I/O Hold Time	30	
TIOV	EXTEST Output Valid Delay		55

XC9500/XL/XV Programming Algorithm

This section describes the programming algorithm executed by the 8051 C-code that reads the XSVF file; this code is contained in the micro.c file in **Appendix C: C-Code Listing, page 31**. This information is valuable to users who want to modify the C-code for porting to other microcontrollers.

The XSVF file contains all XC9500/XL/XV programming instructions and data. This allows the TAP driver code to be very simple. The 8051 interprets the XSVF instructions that describe the CPLD design and then outputs the TAP signals for programming (and testing) the XC9500/XL/XV device. The command sequence for device programming is shown in.

Exception Handling

Figure 19 shows the state diagram for the internal device programming state machine, as defined by the IEEE 1149.1 standard.

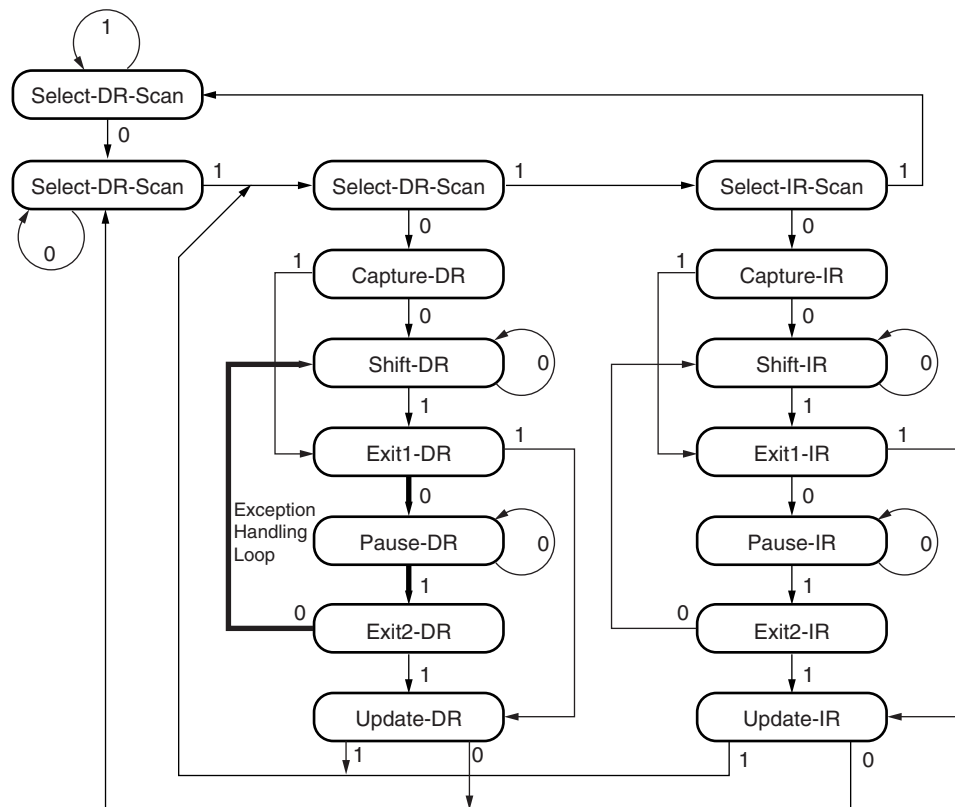


Figure 19: TAP State Machine Flow

Notes:

1. The values shown adjacent to each transition represent the signal present at TMS during the rising edge of TCK.

The C-code drives the 1149.1 TAP controller through the state sequences to load data and instructions, and capture results. One of the key functions performed by the C-code is the TAP controller state transition sequence that is executed when an XC9500/XL/XV program or erase operation needs to be repeated, which can occur on a small percentage of addresses. If a sector or address needs to be reprogrammed or re-erased, the device status bits return a value that is different from that which is predicted in the XSVF file. In order to retry the previous (failed) data, the following 1149.1 TAP state transition sequence is followed, if the TDO mismatch is identified at the EXIT1-DR state:

EXIT1-DR, PAUSE-DR, EXIT2-DR, SHIFT-DR, EXIT1-DR, UPDATE-DR, RUN-TEST/IDLE

The application then increments the previously specified XRUNTEST time by an additional 25 percent and waits for this amount of time in Run-Test/Idle. The effect of this state sequence is to re-apply the previous value rather than apply the new TDI value that was just shifted in.

This “exception handling loop” is attempted no more than N times. If the TDO value does not match after N attempts, the part is defective and a failure is logged. When the retry operation is successful, the algorithm shifts-in the next XSDR data.

The SVF2XSVF **-r** (repeat) option determines the value of N. the recommended value of N for XC9500/XL/XV devices is 16.

XC4000 and Spartan Programming Algorithm

XC4000 Series devices can be configured through the boundary-scan pins. The basic procedure is as follows:

- Power up the FPGA with $\overline{\text{INIT}}$ held Low (or the $\overline{\text{PROGRAM}}$ pin Low for more than 300 ns followed by a High while holding $\overline{\text{INIT}}$ Low). Holding $\overline{\text{INIT}}$ Low allows enough time to issue the CONFIG command to the FPGA. The pin can be used as I/O after configuration if a resistor is used to hold $\overline{\text{INIT}}$ Low
- Issue the CONFIG command to the TMS input
- Wait for $\overline{\text{INIT}}$ to go High
- Sequence the boundary-scan Test Access Port to the SHIFT-DR state
- Toggle TCK to clock data into TDI pin

The user must account for all TCK clock cycles after $\overline{\text{INIT}}$ goes High, as all of these cycles affect the Length Count compare.

For more detailed information, refer to the Xilinx application note XAPP017, “Boundary Scan in XC4000 Devices. This application note also applies to XC4000E and XC4000X devices.

Virtex Programming Algorithm

Virtex devices can be configured through the boundary-scan pins. Configuration through the TAP uses the special CFG_IN instruction. This instruction allows data input on TDI to be converted into data packets for the internal configuration bus.

The following steps are required to configure the FPGA through the boundary-scan port.

- Load the CFG_IN instruction into the boundary-scan instruction register (IR).
- Enter the Shift-DR (SDR) state.
- Shift a standard configuration bitstream into TDI.
- Return to Run-Test-Idle (RTI).
- Load the JSTART instruction into IR.
- Enter the SDR state (For Virtex-II devices, stay in the Run-Test/Idle state).
- Clock TCK for the length of the startup sequence.
- Return to RTI.
- Check the DONE pin status.

See XAPP139: for details on Virtex Configuration.

Notes:

1. The **-fpga** option must be used with the SVF2XSVF translator for Virtex, XC4000, and Spartan devices.
2. The programming operation for each Virtex device ends by checking the DONE pin status. If multiple Virtex devices are to be configured and if the DONE pins of those devices are tied together, then the DONE pin does not go High until all the Virtex devices have been configured. In this case, the check of the DONE pin status for the intermediate Virtex devices fail. To workaround this problem, the check on the DONE pin status for all but the last Virtex device must be removed from the SVF before translation to XSVF.

CoolRunner Programming Algorithm

The CoolRunner devices can be programmed through the boundary-scan pins. The basic procedure is as follows:

- Enter the device into ISP mode
- Erase the entire device
- Program all addresses
- Verify all addresses
- Exit the ISP mode and return to normal functional mode.

XC18V00 PROM Programming Algorithm

The XC18V00 devices can be programmed through the boundary-scan pins. The basic procedure is as follows:

- Enter the device into ISP mode
- Erase the entire device
- Program all addresses
- Apply global operation to refine programmed values.
- Verify all addresses
- Exit the ISP mode and return to normal functional mode.

Conclusion

Xilinx CPLDs and FPGAs are easily programmed by an embedded processor. And, because they are 1149.1 compliant, system and device test functions can also be controlled by the embedded processor, in addition to programming. This capability opens new possibilities for upgrading designs in the field, creating user-specific features, and remote downloading of CPLD/FPGA programs.

Appendix A: SVF File Format for Xilinx Devices

SVF Overview

This appendix describes the Serial Vector Format syntax, as it applies to Xilinx devices; only those commands and command options that apply to Xilinx devices are described. An SVF file is the media for exchanging descriptions of high-level IEEE 1149.1 bus operations which consist of scan operations and movements between different stable states on the 1149.1 state diagram (as shown in [Figure 19](#)). SVF does not explicitly describe the state of the 1149.1 bus at every Test Clock (TCK).

An SVF file contains a set of ASCII statements. Each statement consists of a command and its associated parameters, terminated by a semicolon. SVF is case sensitive, and comments are indicated by an exclamation point (!).

Scan data within a statement is expressed in hexadecimal and is always enclosed in parenthesis. The scan data cannot specify a data string that is larger than the specified bit length; the Most Significant Bit (MSB) zeros in the hex string are not considered when determining the string length. The bit order for scan data defines the LSB (rightmost bit) as the first bit scanned into the device for TDI and SMASK scan data, and is the first bit scanned out for TDO and MASK data.

SVF Commands

The following SVF Commands are supported by the Xilinx devices:

- SDR (Scan Data Register).
- SIR (Scan Instruction Register).
- RUNTEST.

For each of the following command descriptions:

- The parameters are mandatory.
- Optional parameters are enclosed in brackets ([]).
- Variables are shown in *italics*.
- Parenthesis “()” are used to indicate hexadecimal values.
- A scan operation is defined as the execution of an SIR or SDR command and any associated header or trailer commands.

SDR, SIR

```
SDR length TDI (tdi) SMASK (smask)
[TDO (tdo) MASK (mask)];
SIR length TDI (tdi) TDO SMASK (smask);
```

These commands specify a scan pattern to be applied to the target scan registers. The SDR command (Scan Data Register) specifies a data pattern to be scanned into the target device Data Register. The SIR command (Scan Instruction Register) specifies a data pattern to be scanned into the target device Instruction Register.

Prior to scanning the values specified in these commands, the last defined header command (HDR or HIR) is added to the beginning of the SDR or SIR data pattern and the last defined trailer command (TDR or TIR) is appended to the end of the SDR or SIR data pattern.

Parameters:

length — A 32-bit decimal integer specifying the number of bits to be scanned.

[TDI (*tdi*)] — (optional) The value to be scanned into the target, expressed as a hex value. If this parameter is not present, the value of TDI to be scanned into the target device is the TDI value specified in the previous SDR/SIR statement. If a new scan command is specified, which changes the length of the data pattern with respect to a previous scan, the TDI parameter must be specified, otherwise the default TDI pattern is undetermined and is an error.

[TDO (*tdo*)] — (optional) The test values to be compared against the actual values scanned out of the target device, expressed as a hex string. If this parameter is not present, no comparison is performed. If no TDO parameter is present, the MASK is not used.

[MASK (*mask*)] — (optional) The mask to be used when comparing TDO values against the actual values scanned out of the target device, expressed as a hex string. A “0” in a specific bit position indicates a “don’t care” for that position. If this parameter is not present, the mask equals the previously specified MASK value specified for the SIR/SDR statement. If a new scan command is specified which changes the length of the data pattern with respect to a previous scan, the MASK parameter must be specified, otherwise the default MASK pattern is undefined and is an error. If no TDO parameter is present, the MASK is not used.

[SMASK (*smask*)] — (optional) Specifies which TDI data is “don’t care”, expressed as a hex string. A “0” in a specific bit position indicates that the TDI data in that bit position is a “don’t care”. If this parameter is not present, the mask equals the previously specified SMASK value specified for the SDR/SIR statement. If a new scan command is specified which changes the length of the data pattern with respect to a previous scan, the SMASK parameter must be specified, otherwise the default SMASK pattern used is undefined and is an error. The SMASK is used even if the TDI parameter is not present.

Example:

```
SDR 27 TDI (008003fe) SMASK (07ffffff) TDO (00000003) MASK (00000003);
SIR 16 TDO (ABCD);
```

HDR, HIR, TDR, TIR

```
HDR length TDI(tdi) SMASK(smask) [TDO(tdo) MASK(mask)]
HIR length TDI(tdi) SMASK(smask) [TDO(tdo) MASK(mask)]
TDR length TDI(tdi) SMASK(smask) [TDO(tdo) MASK(mask)]
TIR length TDI(tdi) SMASK(smask) [TDO(tdo) MASK(mask)]
```

These commands specify header and trailer bits for data and instruction shifts. Once specified, these bits lead or follow every set of bits shifted for the SIR or SDR commands. These commands are used to specify bits for non-target (bypassed) devices in the scan chain.

The parameters are the same as the SIR and SDR commands.

Example:

```
HDR 1 TDI (0);
TDR 3 TDI (0);
HIR 8 TDI (ff);
TIR 24 TDI (ffffff);
```

RUNTEST

```
RUNTEST run_count TCK;
```

This command forces the target 1149.1 bus to the Run- Test/Idle state for a specific number of microseconds, then moves the target device bus to the IDLE state. This is used to control RUNBIST operations in the target device.

Parameters:

run_count — The number of TCK clock periods that the 1149.1 bus remains in the Run Test/Idle state, expressed as a 32 bit unsigned number.

Example:

```
RUNTEST 1000 TCK;
```

```
! Begin Test Program
TRST OFF;!disable test reset line
ENDIR IDLE;!End IR scan in IDLE
HIR
HDR 16 TDI (FFFF) TDO (FFFF) MASK (FFFF);!16 bit DR Header
TIR
TDR
SIR
SDR
STATE
RUNTEST
```

Figure 20: Sample SVF File

Appendix B: XSVF File Format and Conversion Utilities

This appendix includes the following reference information:

- **The XSVF Commands** — The instructions that are supported, their arguments, and definitions.
- **The svf2xsvf Utility** — Converts the standard SVF file format to the more compact binary XSVF format.

XSVF Commands

The following commands describe the 1149.1 operations in a way that is similar to the SVF syntax. The key difference between SVF and XSVF is that the XSVF file format affords better data compression and therefore produces smaller files.

The format of the XSVF file is a one byte instruction followed by a variable number of arguments (as described in the command descriptions below). The binary (hex) value for each instruction is shown in [Table 4](#):

Table 4: Binary Encoding of XSVF Instructions

XSVF Instruction	Binary Encoding (hex)
XCOMPLETE	0x00
XTDOMASK	0x01
XSIR	0x02
XSDR	0x03
XRUNTEST	0x04
XREPEAT	0x07
XSDRSIZE	0x08
XSDRTDO	0x09
XSETSDRMASKS	0x0a
XSDRINC	0x0b
XSDRB	0x0c
XSDRC	0x0d
XSDRE	0x0e
XSDRTDOB	0x0f
XSDRTDOC	0x10
XSDRTDOE	0x11
XSTATE	0x12
XENDIR	0x13
XENDDR	0x14

XTDOMASK

```
XTDOMASK value<"length" bits>
```

XTDOMASK sets the TDO mask which masks the value of all TDO values from the SDR instructions. Length is defined by the last XSDRSIZE instruction. XTDOMASK can be used multiple times in the XSVF file if the TDO mask changes for various SDR instructions.

Example:

```
XTDOMASK 0x00000003
```

This example defines that TDOMask is 32 bits long and equals 0x00000003

XREPEAT

```
XREPEAT times<1 byte>
```

Defines the number of times that TDO is tested against the expected value before the ISP operation is considered a failure. By default, a device can fail an XSDR instruction 32 times before the ISP operation is terminated as a failure. This instruction is optional.

Example:

```
XREPEAT 0x0f
```

This example sets the command repeat value to 15.

XRUNTEST

```
XRUNTEST time<4 bytes>
```

Defines the amount of time (in microseconds) the device should sit in the Run-Test/Idle state after each visit to the SDR state. The initial XRUNTEST time is zero microseconds.

Notes:

1. For the Virtex-II device, the time parameter must be interpreted as the minimum number of TCK pulses issued within the Run-Test/Idle state after each visit to the SDR state.

Example:

```
XRUNTEST 0x0000fa0
```

This example specifies an idle time of 4000 microseconds.

XSIR

```
XSIR length<1 byte> TDIValue<"length" bits>
```

Go to the Shift-IR state and shift in the TDIValue. If the last XRUNTEST time is non-zero, go to the Run-Test/Idle state and wait for the last specified XRUNTEST time. Otherwise, go to the last specified XENDIR state.

Example:

```
XSIR 0x08 0xec
```

XSDR

```
XSDR TDIValue<"length" bits>
```

Go to the Shift-DR state and shift in TDIValue; compare the TDOExpected value from the last XSDRTDO instruction against the TDO value that was shifted out (use the TDOMask which was generated by the last XTDOMASK instruction). Length comes from the XSDRSIZE instruction.

If the TDO value does not match TDOExpected, perform the exception handling sequence described in the XC9500 programming algorithm section. If TDO is wrong more than the maximum number of times specified by the XREPEAT instruction, then the ISP operation is determined to have failed.

If the last XRUNTEST time is zero, then go to the XENDDR state. Otherwise, go to the Run_Test/Idle state and wait for the XRUNTEST time.

Example:

```
XSDR 02c003fe
```

XSDRSIZE

```
XSDRSIZE length<4 bytes>
```

Specifies the length of all XSDR/XSDRTDO records that follow.

Example:

```
XSDRSIZE 0x0000001b
```

This example defines the length of the following XSDR/XSDRTDO arguments to be 27 bits (4 bytes) in length.

XSDRTDO

```
TDIValue<"length" bits>
TDOExpected<"length" bits>
```

Go to the Shift-DR state and shift in TDIValue; compare the TDOExpected value against the TDO value that was shifted out (use the TDOMask which was generated by the last XTDOMASK instruction). Length comes from the XSDRSIZE instruction.

If the TDO value does not match TDOExpected, perform the exception-handling sequence described in the XC9500 programming algorithm section. If TDO is wrong more than the maximum number of times specified by the XREPEAT instruction, then the ISP operation is determined to have failed.

If the last XRUNTEST time is zero, then go to XENDDR state. Otherwise, go to the Run_Test/Idle state and wait for the XRUNTEST time.

The TDOExpected Value is used in all successive XSDR instructions until the next XSDR instruction is given.

Example:

```
XSDRTDO 0x000007fe 0x00000003
```

For this example, go to the Shift-DR state and shift in 0x000007fe. Perform a logical AND on the TDO shifted out and the TDOMASK from the last XTDOMASK instruction and compare this value to 0x00000003.

XSDRB

```
XSDRB TDIValue<"length" bits>
```

Go to the shift-DR state and shift in the TDI value. Continue to stay in the shift-DR state at the end of the operation. No comparison of TDO value with the last specified TDOExpected is performed.

XSDRC

```
XSDRC TDIValue<"length" bits>
```

Shift in the TDI value. Continue to stay in the shift-DR state at the end of the operation. No comparison of TDO value with the last specified TDOExpected is performed.

XSDRE

```
XSDRE TDIValue<"length" bits>
```

Shift in the TDI value. At the end of the operation, go to the XENDDR state. No comparison of TDO value with the last specified TDOExpected is performed.

XSDRTDOB

```
XSDRTDOB TDIValue<"length" bits> TDOExpected<"length" bits>
```

Go to the shift-DR state and shift in **TDI value**; Compare the TDOExpected value against the TDO value that was shifted out. TDOMask is not applied. All bits of TDO are compared with the TDOExpected. Length comes from the XSDRSIZE instruction.

Because this instruction is primarily meant for FPGAs, if the TDO value does not match TDOExpected, the programming is stopped with an error message. At the end of the operations, continue to stay in the SHIFT-DR state.

XSDRTDOC

```
XSDRTDOC TDIValue<"length" bits>
TDOExpected<"length" bits>
```

Shift in the TDI value; compare the TDOExpected value against the TDO value that was shifted out. Length comes from the XSDRSIZE instruction. TDOMask is not applied. All bits of TDO are compared with the TDOExpected.

If the TDO value does not match TDOExpected, stop the programming operation with an error message. At the end of the operation continue to stay in the SHIFT-DR state.

XSDRTDOE

```
XSDRTDOE TDIValue<"length" bits>
TDOExpected<"length" bits>
```

Shift in the TDI value; compare the TDOExpected value against the TDO value that was shifted out. Length comes from the last XSDSIZE instruction. TDOMask is not applied. All bits of TDO are compared with the TDOExpected.

If the TDO value does not match the TDOExpected, stop the programming operations with an error message. At the end of the operation, go to the XENDDR state.

XSETSDRMASKS

```
XSETSDRMASKS addressMask<"length" bits> dataMask<"length" bits>
```

Set SDR Address and Data Masks. The address and data mask of future XSDRINC instructions are indicated using the XSETSDRMASKS instructions. The bits that are 1 in addressMask indicate the address bits of the XSDR instruction; those that are 1 in dataMask indicate the data bits of the XSDR instruction. "Length" comes from the value of the last XSDRSize instruction.

Example:

```
XSETSDRMASKS 00800000 000003fc
```

XSDRINC

```
XSDRINC startAddress<"length" bits>
numTimes<1 byte> data[1]<"length2" bits> ...data[numTimes]<"length2" bits>
```

Do successive XSDR instructions. Length is specified by the last XSDRSize instruction. Length2 is specified as the number of 1 bits in the dataMask section of the last XSETSDRMASKS instruction.

The startAddress is the first XSDR to be read in. For numTimes iterations, increment the address portion (indicated by the addressMask section of the last XSETSDRMASKS instruction) by 1, and load in the next data portion into the dataMask section.

Notes:

1. An XSDRINC <start> 255 data0 data1 ... data255 actually does 256 SDR instruction since the start address also represents an SDR instruction.

Example:

```
XSDRINC 004003fe 05 ff ff ff ff ff
```

XCOMPLETE

```
XCOMPLETE
```

End of XSVF file reached.

Example:

```
XCOMPLETE
```

XSTATE

```
xstate state <1 byte>
```

If state is zero, force TAP to Test-Logic-Reset state by holding TMS High and applying 5 TCK cycles. If state is one, transition TAP from Test-Logic-Reset to Run-Test/Idle.

XENDIR

```
xendir state <1 byte>
```

Set the XSIR end state to Run-Test/Idle (0) or Pause-IR (1). The default is Run-Test/Idle.

XENDDR

```
XENDDR state <1 byte>
```

Set the XSDR and XSDRTDO end state to Run-Test/Idle (0) or Pause-DR (1). The default is Run-Test/Idle.

svf2xsvf File Conversion Utility

This executable reads in an SVF file (generated by JTAG Programmer) and generates an XSVF file.

Usage:

```
svf2xsvf [-d] [-fpga] [-rlen number] [-r number] [-extensions] -i<svf file>  
-o<svf file> -a<text file>
```

Options:

- d** — delete pre-existing output files.
- r *number*** — Set the XREPEAT value to number
- fpga** — FPGA device
- rlen** — Create records of length specified by **rlen** (FPGA only)
- extension** —use XENDIR and XENDDR.

Example for XC9500/XL/XV:

```
svf2xsvf -i file.svf -o file.xsvf -a file.txt
```

Example for XCV1800:

```
svf2xsvf -i file.svf -o file.xsvf -a file.txt
```

Example for CoolRunner:

```
svf2xsvf -extensions -r 0 -i file.svf -o file.xsvf -a file.txt
```

Example for FPGA:

```
svf2xsvf -fpga -i file.svf -o file.xsvf -a file.txt
```

mergexsvf File Merge Utility

This executable takes multiple XSVF files and merges them into a single XSVF file. When the files are merged, the XCOMPLETE commands are removed from the intermediate file images and a header is inserted between files that resets the parameters for the following commands: XSTATE, XENDIR, XENDDR, and XRUNTEST.

Usage:

```
mergexsvf [-d] [-v2] -o <output.xsvf> -i <input1.xsvf> -i <input2.xsvf> [-  
i <inputN.xsvf>...]
```

Options:

- d** – Delete pre-existing output file.
- i <inputN.xsvf>** – Input files to be merged in the order listed.
- o <output.xsvf>** – Merged output file.
- v2** – Generates an output file with intermediate headers that do not include the XSTATE, XENDIR, and XENDDR commands.

Notes:

1. The input XSVF files should be generated using the -v2 option with the svf2xsvf file conversion utility.

Example:

```
mergexsvf -d -o merged.xsvf -i xc9536xl.xsvf -i xc18v04.xsvf
```

Appendix C: C-Code Listing

The following files contain the C source code used to read an XSVF file and output the appropriate Test Access Port control bits:

C-Code Files

- **lenval.c** — This file contains routines for using the lenVal data structure.
- **micro.c** — This file contains the main function call for reading in a file from an EPROM and driving the JTAG signals.
- **ports.c** — This file contains the routines to output values on the JTAG ports, to read the TDO bit, and to read a byte of data from the EPROM.

Header Files

- **lenval.h** — This file contains a definition of the lenVal data structure and extern procedure declarations for manipulating objects of type lenVal. The lenVal structure is a byte oriented type used to store an arbitrary length binary value.
- **ports.h** — This file contains extern declarations for providing stimulus to the JTAG ports.

To compile this C-code for a microcontroller, only four functions within the ports.c file need to be modified:

- **setPort** — Sets a specific port on the microcontroller to a specified value.
- **readTDOBit** — Reads the TDO port.
- **readByte** — Reads a byte of data from the XSVF file.
- **waitTime** — Pauses for a specified amount of time.

Notes:

1. The waitTime function is called when the device is in Run-Test/Idle state to pause the system for the specified amount of time. For all device families other than the Virtex-II, TCK pulses are not required (but can occur) while the waitTime function is pausing the system. For the Virtex-II devices, the parameter to the waitTime function must be interpreted as a minimum number of TCK pulses to be generated. Typically, the Virtex-II devices require less than 25 TCK pulses.

The following is an example implementation of the waitTime function that is sufficient for all Xilinx devices except for Virtex-II devices. (If you have a more accurate timing function than the standard sleep function, use it for improved programming performance.)

```
void waitTime( long microseconds )
{
    // Round up to the nearest millisecond
    sleep( ( microseconds + 999L ) / 1000L );
}
```

The following are code examples for implementations of the waitTime function that handle all the device families including the Virtex-II. For systems that can clock TCK at 1 MHz or faster, the waitTime function can be implemented so that it generates TCK pulses equivalent to the requested wait time:

```
void waitTime( long microseconds )
{
    // tckCyclesPerMicrosecond is a predetermined constant for your system
    long tckPulses = microseconds * tckCyclesPerMicrosecond;
    for ( long i = 0; i < tckPulses; ++i )
    {
        pulseTCK();
    }
}
```

For systems that can run TCK significantly slower than 1 MHz, the waitTime implementation needs to satisfy the Virtex-II requirement for the few TCK cycles (<25) that it needs, but the implementation should consider optimizing the wait time for the longer wait periods that other devices require:

```
void waitTime( long microseconds )
{
    if ( microseconds >= 50 )
    {
        // Round up to the nearest millisecond and
        //use standard sleep function
        sleep( ( microseconds + 999L ) / 1000L );
    }
    else // satisfy Virtex-II TCK clock cycles
    {
        for ( long i = 0; i < microseconds; ++i )
        {
            pulseTCK();
        }
    }
}
```

For help in debugging the code, a compiler switch called DEBUG_MODE is provided. If DEBUG_MODE is defined, the software reads from an XSVF file (which must be named prom.bit) and prints the debugging messages. A compile switch called XSVFSIM allows the designer to simulate the TAP outputs without a physical connection to the target device. Uses the DEBUG_MODE with the XSVFSIM switch to view the simulated TAP signal values.

Appendix D: Dynamically Selecting Target Devices for Configuration

In the default configuration flow, the complete JTAG scan chain is defined in the JTAG Programmer software. Designs are assigned to devices within the JTAG scan chain, and the devices to be configured are selected prior to the creation of the SVF file. The devices selected for configuration are called target devices. JTAG Programmer generates an SVF file that contains a separate set of configuration commands and data for each target device. Target devices are configured sequentially, one device at a time. When a target device gets configured, the non-target devices are put into bypass mode. Each set of SVF commands and data for a target device contains an exact complement of bits corresponding to the bypassed, non-target devices. Thus, the exact assignment of designs and exact selection of target devices must be known in advance, because each SVF is built for a specific scan chain and specific selection of target devices.

The default configuration flow is inefficient for systems that use identical designs on multiple FPGAs or that use multiple combinations of designs for a set of FPGAs. For systems that configure multiple FPGAs with the same design, the SVF must still be created with separate sets of commands and data for each FPGA. That is, the design data is duplicated for each FPGA to be configured. For systems that use multiple combinations of designs across a set of FPGAs, SVF files must exist for each possible combination of design assignments. Again, design data is duplicated within the system. Because a one-to-one correspondence exists between the original SVF file and the corresponding XSVF file used in the embedded environment, the creating of inefficient SVF files equivalently affects the XSVF file storage requirements.

Using Dynamic Targeting to Reduce System Storage Requirements

To improve the data storage efficiency of these particular systems, a special version of the XSVF player is included in the XAPP058 download package. This special version of the XSVF player uses XSVF files built to configure just one device and supports the ability to dynamically target a given XSVF file to configure any compatible device in the scan chain. Only one XSVF file per design is required. In a system that uses identical designs on multiple FPGAs, a single XSVF (design) file can be reused to configure all of the FPGAs. In a system that uses multiple combinations of designs for a set of FPGAs, separate XSVF files corresponding to each design can be dynamically selected and targeted to the FPGAs.

The dynamic targeting feature reduces system storage requirements in the following systems:

- Systems in which FPGAs are configured with identical designs
- Systems in which a set of FPGAs can be configured with multiple combinations of selected designs.

C-Code Files for the Dynamic Targeting XSVF Player

The files for this special version of the XSVF player are located in the `dynamic_target` directory from the download package. The `dynamic_target` directory contains two files: `micro_dynamic_target.c` and `micro_dynamic_target.h`. These two files are modified versions of the base `micro.c` and `micro.h` source files from the `src` directory in the download package. The code in the `micro_dynamic_target.c` file has been modified to support dynamic selection of the device to be configured within a scan chain. The `micro_dynamic_target.h` file simply contains the declaration of the modified procedural interface that supports this dynamic targeting feature.

Substitute the `dynamic_target` files for the base `micro.c` and `micro.h` files in `src` directory to build an XSVF player that supports the dynamic targeting feature:

- Copy `dynamic_target\micro_dynamic_target.h` `src\micro.h`
- Copy `dynamic_target\micro_dynamic_target.c` `src\micro.c`

Building SVF (XSVF) Files for Dynamic Targeting

An XSVF file that is used to configure a dynamically selected device at run-time must contain just the set of commands and data to configure a single, compatible device.

To create an SVF file for dynamic targeting, use JTAG Programmer to:

1. Define a scan chain that contains just the single device.
2. Assign the design file to the device in the scan chain.
3. Select the device as the operation target.
4. Generate the SVF file that contains the program operation commands and data for the assigned design.

Finally, the XSVF file must be created from the SVF file using the SVF2XSVF translator.

A separate XSVF file must be created for each design used to configure a device. These XSVF files are individually used to configure selected devices in the system.

A Primer on the Dynamic Targeting Feature

The basic commands within an XSVF file are designed to shift instruction and data bits through the JTAG scan chain into a target device. The commands in an XSVF file built for a single-device scan chain effectively shift the instruction and data bits directly into the JTAG ports of the target device. To dynamically retarget a single-device XSVF file to a specific device in a multi-device scan chain, the XSVF player must account for the shift registers of the non-target devices in the scan chain and insert the appropriate bits before or after the target device's instruction or data bit sets.

The IEEE Standard 1149.1 specifies the BYPASS instruction to consist of all one-bits and the BYPASS data register to be exactly one-bit wide. With this information, the exact bit patterns for the bypassed, non-target devices can be calculated. During an instruction shift, one-bits must be shifted into the instruction registers of all the bypassed, non-target devices. During a data shift, an extra data (zero) bit must be shifted into the bypass registers of all non-target devices.

Using the Special XSVF Player to Dynamically Select Target Devices

In the regular XSVF player, a pointer to the beginning of the XSVF data is first set. Then, the start function (`xsvfExecute`) is called to execute the XSVF data. The same flow applies to the special XSVF player with additional parameters that must be specified to the start function.

The primary function (`xsvfExecute`) that starts the special `dynamic_target` XSVF player has been enhanced with five additional parameters. These parameters specify the number of leading and trailing instruction and data bits to be inserted before or after the main set of bits from the XSVF commands. An additional parameter is accepted that aligns Virtex configuration data to a 32-bit boundary. (See XAPP139 for additional information on the Virtex 32-bit configuration frame that imposes the 32-bit boundary requirement on the bitstream.)

The enhanced `xsvfExecute` function is declared in the `micro_dynamic_target.h` file as follows:

```
int xsvfExecute(int iHir, int iTir, int iHdr, int iTdr, int iHdrFpga);
```

The parameters are described in [Table 5](#).

Notes:

1. The 32-bit alignment issue applies only to the Virtex, Virtex-E, and Spartan-II device families.

Table 5: XSVF Player Parameters

Parameter	Name	Description
iHir	Header Instruction Register	The number of (one) bits to shift before the target set of instruction bits. These bits put the non-target devices after the target device into BYPASS mode. The iHir value must be equivalent to the sum of instruction register lengths for devices following the target device in the scan chain.
iTir	Trailer Instruction Register	The number of (one) bits to shift after the target set of instruction bits. These bits put the non-target devices before the target device into BYPASS mode. The iTir value must be equivalent to the sum of instruction register lengths for devices preceding the target device in the scan chain.
iHdr	Header Data Register	The number of (zero) bits to shift before the target set of data bits. These bits are placeholders that fill the BYPASS data registers in the non-target devices after the target device. The iHdr value must be equivalent to the sum of devices following the target device in the scan chain.
iTdr	Trailer Data Register	The number of (zero) bits to shift after the target set of data bits. These bits are placeholders that fill the BYPASS data registers in the non-target devices before the target device. The iTdr value must be equivalent to the sum of devices preceding the target device in the scan chain.
iHdrFpga	Header Data Register for the Virtex FPGA Commands	The number of (zero) bits to shift before the target set of Virtex FPGA data bits. These bits are used to align the configuration bitstream for Virtex devices to a 32-bit boundary. The iHdrFpga value must be equivalent to 32 minus the sum of devices preceding the target device in the scan chain. If no devices precede the target device, the value is zero. If the sum of devices is greater than 32, then the value must be 32 minus the modulo [32] of the sum of devices preceding the target device.

These parameters are equivalent to the HIR, TIR, HDR, and TDR commands in the SVF specification. See the SVF specification for further details:

<http://support.asset-intertech.com/svf.htm>

From the given set of parameters, the `micro_dynamic_target.c` implementation automatically adds the necessary set of complementary bits to the XSVF commands to compensate for the bypassed devices in the scan chain.

Notes:

If all of the `xsvfExecute` parameters are equal to zero, then the special XSVF player functionality is equivalent to the base XSVF player that takes an XSVF file created for a fully specified scan chain! Thus, the special XSVF player with the dynamic targeting feature can be used in both the normal (fully-specified XSVF) and special (dynamic targeting) modes.

Dynamic Target Example

To configuring four Virtex 300E devices with identical designs using a single XSVF source file, the original SVF file must be created using the instructions from the “Building SVF (XSVF) Files for Dynamic Targeting” section. Assuming the design for an XCV300E is located in the design.bit file, the XSVF file must be created as follows:

1. Define a scan chain in JTAG Programmer with just the single XCV300E device.
2. Assign the design.bit file to the single instance of the XCV300E in the scan chain.
3. Select the XCV300E as the operation target.
4. Generate an SVF to program the XCV300E.
5. Translate the SVF to XSVF using the SVF2XSVF translator with the **-fpga** option.

In the embedded environment,

1. Reset the XSVF program pointers to point to the beginning of the XSVF data.
2. To program device #1, call the xsvfExecute function with the following parameters:
xsvfExecute(15, 0, 3, 0, 0)
3. Reset the XSVF program pointers to point to the beginning of the XSVF data.
4. To program device #2, call the xsvfExecute function with the following parameters:
xsvfExecute(10, 5, 2, 1, 31)
5. Reset the XSVF program pointers to point to the beginning of the XSVF data.
6. To program device #3, call the xsvfExecute function with the following parameters:
xsvfExecute(5, 10, 1, 2, 30)
7. Reset the XSVF program pointers to point to the beginning of the XSVF data.
8. To program device #4, call the xsvfExecute function with the following parameters:
xsvfExecute(0, 15, 0, 3, 29)

Further examples of the code for the four device Virtex scan chain and a four device XC18V00 scan chain can be found in the dynamic_target directory of the download package.

An example XSVF player executable that provides this dynamic targeting capability is available under the **playxsvf\Release_DT** directory. This executable runs on a Windows 95/98/Me/NT/2000 PC with the Xilinx Parallel Cable III.

Appendix E: Binary to Intel Hex Translator

This appendix contains C-code that can be used to convert XSVF files to Intel Hex format for downloading to an EPROM programmer. Most embedded processor code development systems can output Intel Hex for included binary files, and for those systems the following code is not needed. However, designers can use the following C-code if the development system they are using does not have Intel Hex output capability.

Overview

The ISP controller described in this application note allows designers to program and test XC9500/XL CPLDs from information stored in EPROM. This information is saved in a binary XSVF file that contains both device programming instructions as well as the device configuration data. The 8051 microcontroller reads the EPROM (or EPROMs) that contain the XSVF file, converts the binary information to XC9500/XL compatible instructions and data, and outputs the programming information to the XC9500/XL device through a 4-wire test access port.

After an XC9500/XL design has been converted to XSVF format, the XSVF information is converted to Intel hex format which is downloaded to an EPROM programmer. The resulting EPROMs, containing the CPLD programming information, can then be used in this ISP controller design.

```

/*
   This program is released to the public domain. It
   prints a file to stdout in Intel HEX 83 format.
*/

#include <stdio.h>

#define RECORD_SIZE0x10/* Size of a record. */
#define BUFFER_SIZE 128

/** Local Global Variables */

static char *line, buffer[BUFFER_SIZE];
static FILE *infile;

/** Extern Functions Declarations */

extern char hex( int c );
extern void puthex( int val, int digits );

/** Program Main */

main( int argc, char *argv[] )
{
    int c=1, address=0;
    int sum, i;
    i=0;
    /** First argument - Binary input file */

    if (!(infile = fopen(argv[++i],"rb"))) {
        fprintf(stderr, "Error on open of file %s\n",argv[i]);
        exit(1);
    }

    /** Read the file character by character */

    while (c != EOF) {
        sum = 0;
        line = buffer;
        for (i=0; i<RECORD_SIZE && (c=getc(infile)) != EOF; i++) {

```

```

        *line++ = hex(c>>4);
        *line++ = hex(c);
        sum += c; /* Checksum each character. */
    }
    if (i) {
        sum += address >> 8; /* Checksum high address byte.*/
        sum += address & 0xff; /* Checksum low address byte.*/
        sum += i;          /* Checksum record byte count.*/
        line = buffer;    /* Now output the line! */
        putchar(':');
        puthex(i,2);      /* Byte count. */
        puthex(address,4); /* Do address and increment */
        address += i;     /* by bytes in record. */
        puthex(0,2);     /* Record type. */
        for(i*=2;i;i--)  /* Then the actual data. */
            putchar(*line++);
        puthex(0-sum,2); /* Checksum is 1 byte 2's comp.*/
        printf("\n");
    }
}

printf(":00000001FF\n"); /* End record. */
}

/* Return ASCII hex character for binary value. */

char
hex( int c )
{
    if((c &= 0x000f)<10)
        c += '0';
    else
        c += 'A'-10;
    return((char) c);
}

/* Put specified number of digits in ASCII hex. */

void
puthex( int val, int digits )
{
    if (--digits)
        puthex(val>>4,digits);
    putchar(hex(val & 0x0f));
}

```

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/15/01	3.0	Revised Xilinx release.
06/25/04	3.1	Minor changes made.